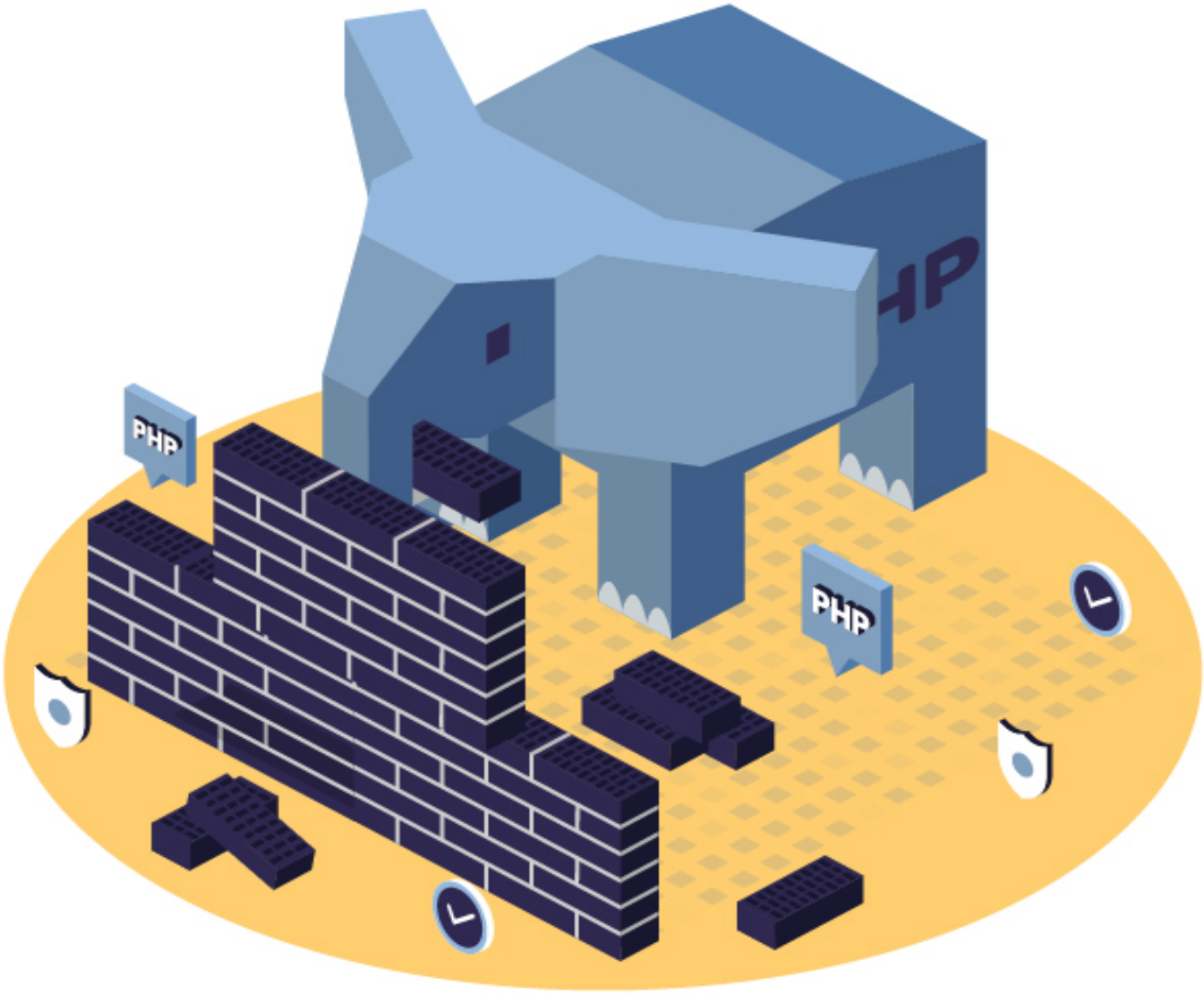
Programação Web avançada com PHP

Construindo software com componentes



Casa do Código | alura

FLÁVIO LISBOA

Sumário

- ISBN
- Agradecimentos
- Sobre o autor
- Prefácio
- 1. Introdução
- 2. PHP para quem tem pressa
- 3. A aplicação de exemplo
- 4. Desenvolvimento orientado a componentes
- 5. Desenvolvimento orientado a eventos
- 6. Injeção de dependências
- 7. Segurança de aplicações web
- 8. Filtros e conversores de dados
- 9. Validadores de dados
- 10. Autenticação
- 11. Controle de permissões
- 12. Mapeamento objeto-relacional com Laminas\Db
- 13. Web services e APIs
- 14. Lendo e gravando dados de configuração
- 15. Internacionalização
- 16. Referências

ISBN

Impresso e PDF: 978-65-86110-25-8

EPUB: 978-65-86110-23-4

MOBI: 978-65-86110-24-1

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Agradeço a Deus, pelos milagres e por me sustentar na dificuldade.

Agradeço à minha esposa, por ser minha companheira.

Agradeço à minha filha, por ser mais do que eu sonhava quando a imaginei.

Agradeço a toda a equipe da Casa do Código, por tornar o projeto deste livro realidade, principalmente à Vivian, que transformou o caos do meu pensamento em algo legível.

Agradeço aos meus colegas da comunidade PHP, que me ensinaram muito e mostraram caminhos para trabalhar melhor. E a todos os leitores que enviaram perguntas, críticas, sugestões e correções.

Agradeço aos alunos que tenho ensinado e treinado há mais de uma década, os quais compartilharam seus anseios e realidades, e ampliaram meu horizonte.

Agradeço aos artistas de histórias em quadrinhos, por estimularem minha imaginação e mostrarem que é possível aprender com diversão.

Agradeço aos excelentes professores que tive a honra de conhecer.

Agradeço à minha mãe, onde quer que ela esteja, e aos que duvidaram do que eu era capaz de fazer.

Sobre o autor

Flávio Gomes da Silva Lisboa (FGSL) é mestre e doutor em Tecnologia e Sociedade pela Universidade Tecnológica Federal do Paraná e bacharel em Ciência da Computação, com especialização em Tecnologia Java. Programador formado pelo Centro Estadual de Educação Tecnológica Paula Souza, já atuou em empresas privadas de TI e foi funcionário do Banco do Brasil, onde atuou como analista na diretoria internacional. É analista de desenvolvimento do Serviço Federal de Processamento de Dados (Serpro), no qual foi coordenador do *Programa Serpro de Software Livre* e gerente de equipe de desenvolvimento.

Tem mais de 10 anos de experiência em treinamento de desenvolvedores em programação orientada a objetos, padrões de projeto e uso de frameworks. Lecionou na pós-graduação da UNICID, em São Paulo, nas faculdades Sant'Anna, em Ponta Grossa, na UNICESUMAR, em Maringá, e na Faculdade Alfa de Umuarama. Lecionou no curso de sistemas de informação do Isulpar, em Paranaguá. Atualmente é professor de ensino médio e técnico da ETEC Bento Quirino, do Centro Estadual de Educação Tecnológica Paula Souza. É pioneiro na bibliografia em língua portuguesa sobre Zend Framework e Symfony. Por fim, ele é associado a *ABRAPHP*, *Zend PHP Certified Engineer*, *Zend Framework Certified Engineer* e *Zend Framework 2 Certified Architect*.

É autor do documentário em seis volumes *Rom, Biografia não Autorizada*, e dos livros de ficção *O Um: a solidão e a harmonia* e *Pandino, o Imperador, a vingança do Um*, estes dois últimos reunidos no volume único *Livres ou Iguais: a vingança de Atsu Col, o Um Que é Todos, contra Pandino, o Imperador*. Também é pesquisador, concentrando seus estudos no tema do software livre e aberto. Seus artigos podem ser encontrados na Biblioteca Digital da Sociedade Brasileira de Computação, a SBC-OpenLib (SOL).

Prefácio

Este livro é para o(a) desenvolvedor(a), ou candidato(a) a desenvolvedor(a), que provavelmente já consumiu o material introdutório ofertado sobre a linguagem de programação PHP pela infinidade de cursos básicos disponíveis na Internet, mas que ainda não conseguiu dar o próximo passo para lidar com a complexa coordenação de várias responsabilidades em uma aplicação web.

Há muitos livros disponíveis sobre fundamentos de programação em PHP e também há livros que ensinam a operar frameworks, mas vários pecam por não mostrar que quem programa tem controle sobre os componentes do framework, que pode modificá-los e que o framework não é a linguagem de programação.

Principalmente em um cenário de microsserviços, é preciso focar em usar componentes de forma desacoplada, sem uma estrutura full stack que controla o programador, quando é o programador quem deveria controlar o programa. Falta algo que avance em complexidade e que seja prático.

A proposta deste livro é abordar tópicos avançados de programação PHP orientada a objetos para aplicações web. A ideia é apresentar uma aplicação iniciada e refatorá-la a cada capítulo, mostrando como usar componentes específicos de software - e entendendo como eles funcionam.

O conteúdo foi organizado da seguinte forma: no capítulo 1, fazemos uma introdução ao ambiente de desenvolvimento necessário para a execução dos exemplos que são construídos ao longo do livro. Utilizaremos a versão 8.2 da linguagem PHP, a versão 10.4 do gerenciador de banco dados MariaDB (compatível com MySQL) e a versão 1.9.5 do ambiente integrado Visual Studio Code.

O capítulo 2 tem dupla finalidade. Ele pode funcionar como uma revisão, para quem já tem uma base de conhecimento em PHP, mas precisa lembrar de alguns detalhes para lidar com um framework.

Ou ele pode funcionar como um curso básico rápido, para quem não tem conhecimento específico sobre a linguagem, mas se acha em condições de assimilar rapidamente os conceitos introdutórios. Dentro do livro ele é uma exceção, pois pode ser pulado caso o leitor sinta-se seguro com relação à Programação Orientada a Objetos usando padrões de projeto implementados por um framework.

Tendo certeza de que você tem os fundamentos necessários, no capítulo 3, apresentamos e descrevemos a aplicação de exemplo, que será utilizada para a implementação dos conceitos apresentados nos capítulos posteriores. Nessa aplicação já estão implementados conceitos que consideramos como fundamentais para o desenvolvimento de aplicações web com a linguagem de programação PHP.

No capítulo 4, utilizamos a aplicação de exemplo para mostrar de forma prática como se aplica o paradigma do desenvolvimento orientado a componentes.

No capítulo 5, utilizamos a mesma aplicação para mostrar outro paradigma, o desenvolvimento orientado a eventos.

No capítulo 6, mostramos como a aplicação de exemplo utiliza a técnica de injeção de dependências.

Após verificar o que a aplicação de exemplo já possui, passamos a abordar o que falta nela. O primeiro conjunto de necessidades, abordado de forma geral no capítulo 7, é a segurança de uma aplicação web - a segurança que ela própria deve prover, independentemente da infraestrutura na qual está instalada. Nos capítulos de 8 a 11, nos aprofundamos em tópicos específicos de segurança: filtros e conversores de dados, validadores de dados, autenticação e controle de permissões.

No capítulo 12, abordamos alternativas de implementação de mapeamento objeto-relacional, questionando as vantagens e desvantagens de cada uma.

No capítulo 13, tratamos de um assunto extremamente relevante para uma realidade de aplicações baseadas na concepção de microsserviços, que é a construção de web services, os quais são as bases para a criação de APIs.

No capítulo 14, abordamos a leitura e gravação de dados de configuração.

No capítulo 15, finalizamos com a implementação da internacionalização da aplicação, que é a capacidade de traduzir os textos de sua interface com o usuário para qualquer idioma.

Todo código-fonte dos projetos deste livro está disponível em https://github.com/fgsl/phpbooksample.

Boa sorte, e que o PHP esteja com você!

Capítulo 1
Introdução

O confronto com a dificuldade da programação quase causou um grave rombo na fé de que os computadores seriam maravilhosos. – Edsger W. Dijkstra

Neste capítulo inicial, prepararemos o ambiente para a execução dos exemplos que serão apresentados a partir do próximo capítulo. Precisamos criar um ambiente em que seja possível executar um programa escrito com a linguagem de programação PHP a partir de um servidor Web e estabelecer conexão com um banco de dados relacional. Descreveremos a seguir como criar esse ambiente e proporemos um editor de código para a construção dos exemplos. Todos os softwares mencionados neste capítulo são livres e não exigem pagamento de licenças para seu uso.

1.1 PHP e MySQL

Desde a versão 5.4, a linguagem de programação PHP possui um servidor Web embutido, que provê as funcionalidades necessárias para um ambiente de desenvolvimento. Isso torna desnecessária a instalação de um servidor Apache ou Nginx para a reprodução dos exemplos deste livro.

Adotaremos o XAMPP como ambiente de desenvolvimento, para termos PHP e um banco de dados relacional disponíveis, sem nos preocuparmos com peculiaridades do sistema operacional utilizado.

Trata-se de um pacote de softwares que inclui principalmente Apache, MariaDB (compatível com MySQL), PHP e PEAR (o X refere-se ao sistema operacional, e as demais letras são iniciais desses

softwares). O site do projeto é: https://www.apachefriends.org/pt_br/index.html/.

O XAMPP possui distribuições para GNU/Linux, Mac OS X e Windows. Ele é o principal produto da Apache Friends, uma organização sem fins lucrativos, criada para promover o uso do servidor web Apache.

Na página do XAMPP, mencionada anteriormente, clique no botão de `Download` correspondente ao seu sistema operacional. Neste livro, foi utilizada uma instalação de XAMPP com PHP 8.2. Você pode instalar exatamente essa versão ou uma superior, se estiver disponível. Versões anteriores do XAMPP podem ser obtidas em https://www.apachefriends.org/download.html/.

Ao clicar no botão de `Download`, será baixado um instalador. Execute-o em sua máquina e ele abrirá uma interface gráfica que o conduzirá na instalação. XAMPP dispõe de um painel de controle gráfico que permite iniciar e parar os serviços do Apache e MySQL. Como usaremos o servidor embutido do PHP, basta iniciar o MySQL. A seguir, apresentamos a sequência de telas da instalação do XAMPP.

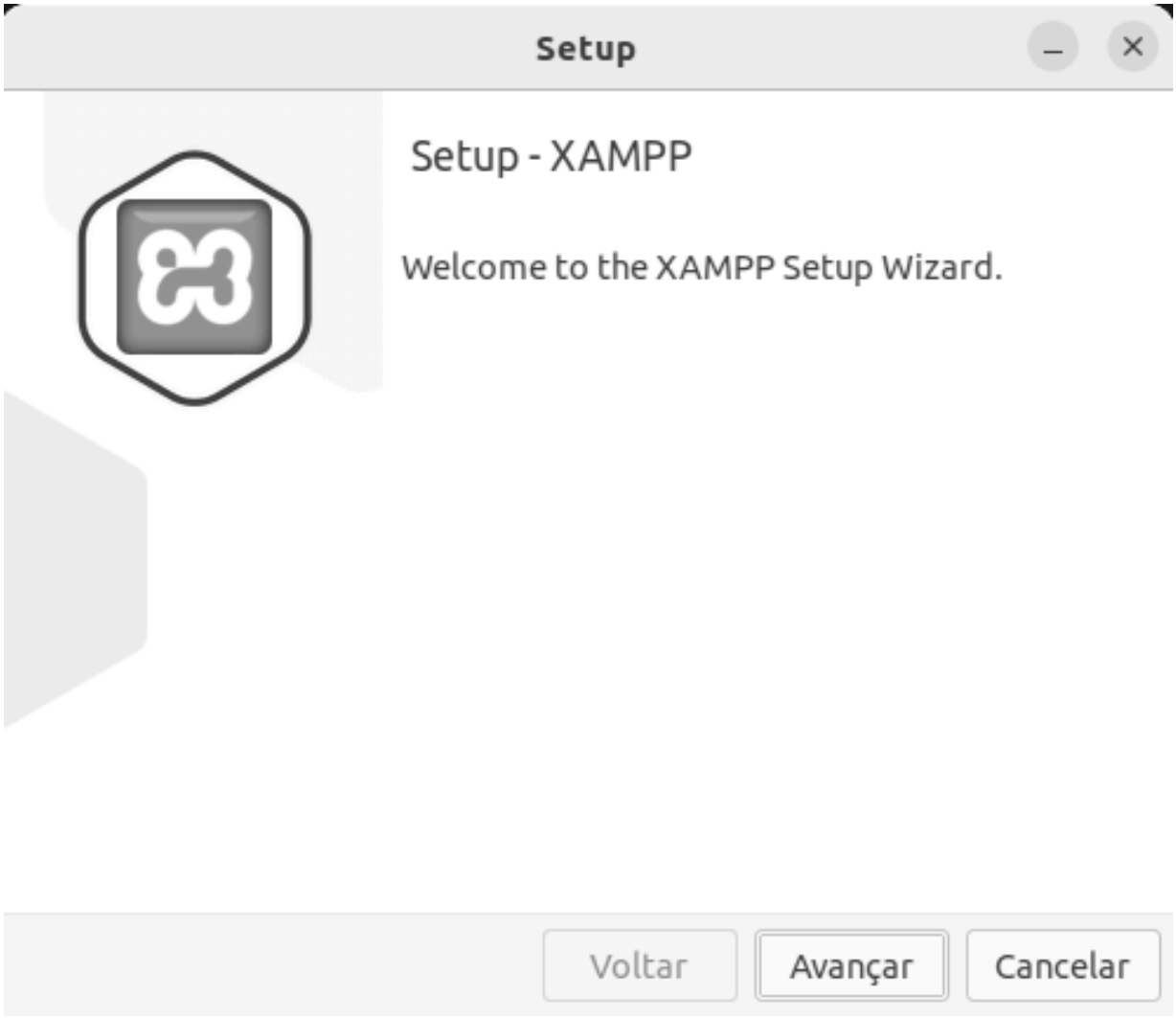


Figura 1.1: Tela inicial do instalador do XAMPP

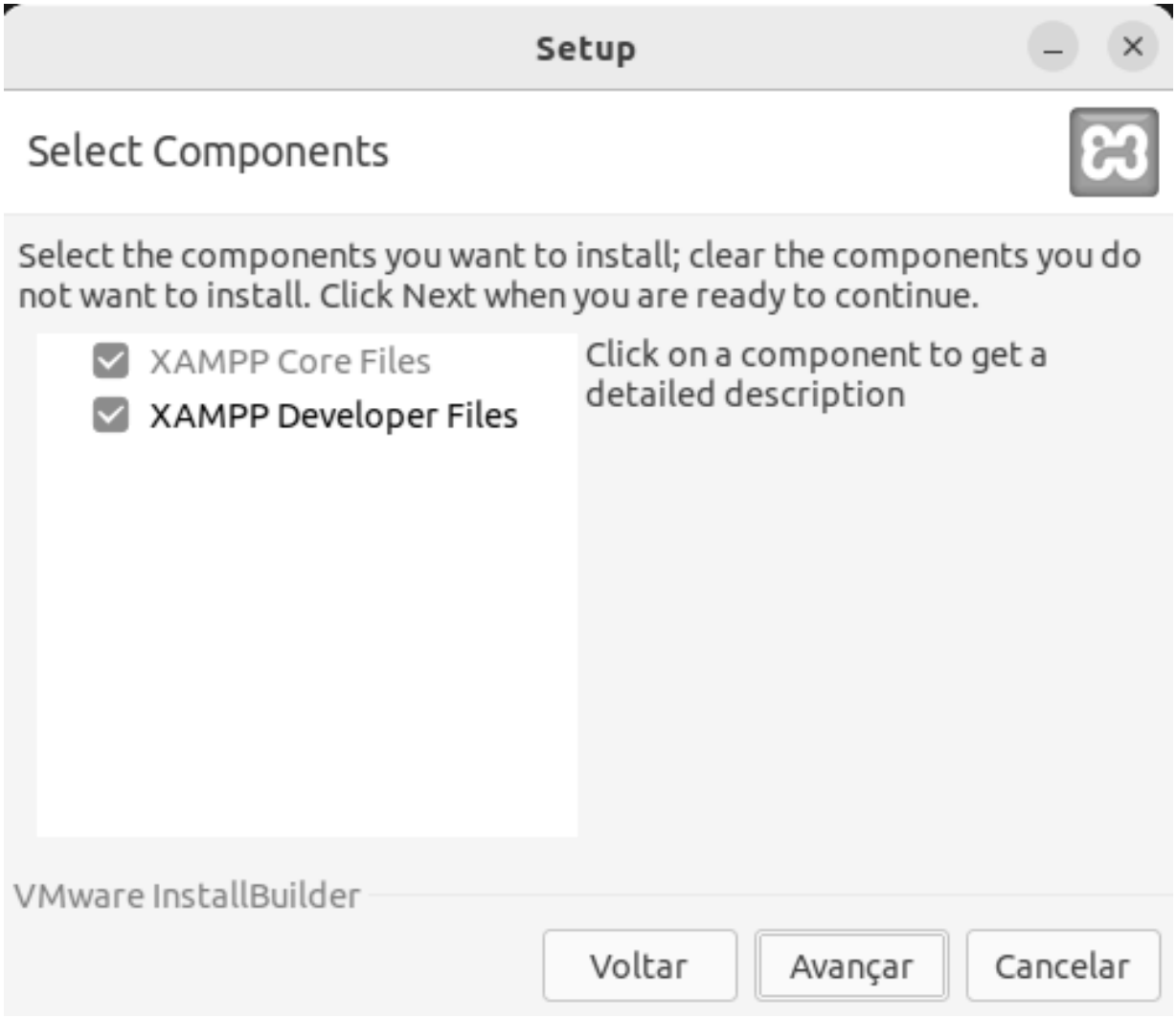


Figura 1.2: Seleção de componentes do XAMPP a serem instalados

A partir da tela anterior, basta prosseguir clicando em `Next` até surgir o botão `Avançar` (ou `Finish`) . Clique nele para ver o painel de controle:

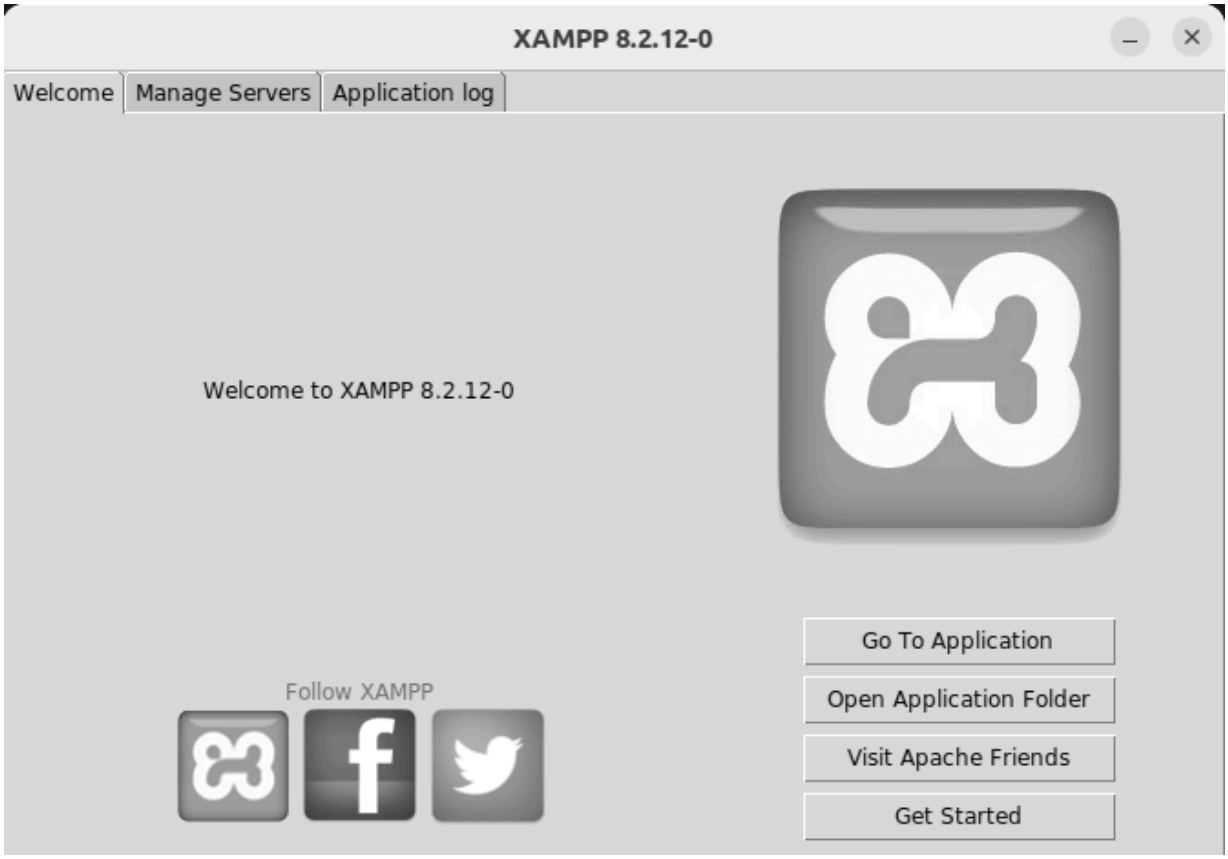


Figura 1.3: Aba Welcome do painel de controle do XAMPP

Na aba *Manage Servers* podemos iniciar o serviço do gerenciador de banco de dados MySQL, conforme mostra a próxima figura, selecionando o item *MySQL Database* e clicando no botão `Start`.

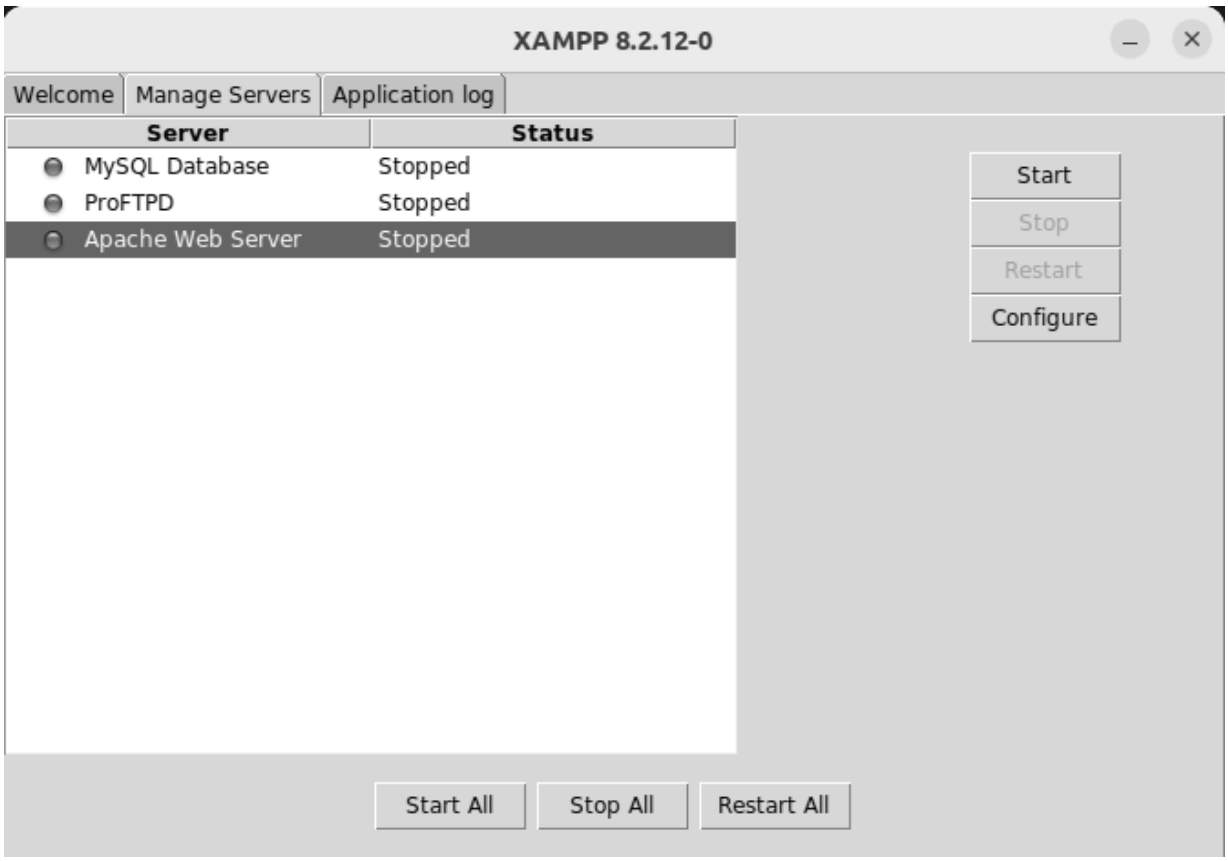


Figura 1.4: Aba de gerenciamento dos serviços do XAMPP

1.2 Ambiente integrado de desenvolvimento

PHP não depende de nenhum editor de código específico. Neste livro utilizaremos Visual Studio Code como ambiente integrado de desenvolvimento. Vamos chamá-lo daqui em diante de VSCode apenas.

Se você não tiver o VSCode instalado, acesse https://code.visualstudio.com/ e baixe o instalador para seu sistema operacional.

Pronto, já temos as ferramentas instaladas e configuramos nosso ambiente. No próximo capítulo, faremos uma revisão (ou uma rápida

introdução) para nos prepararmos para a construção de uma aplicação PHP com recursos avançados de programação.

O capítulo a seguir é excepcional dentro deste livro. Se você se sentir preparado para trabalhar com uma aplicação PHP orientada a objetos usando padrões de projeto implementados por um framework, pode pulá-lo. Mas se não se sentir seguro, ou, se quiser fazer uma revisão a qualquer momento, encontrará nele uma referência dos fundamentos da linguagem PHP necessários para dominar tópicos avançados.

Você pode fazer a experiência de começar a ler o capítulo 3 direto. Se sentir qualquer desconforto com a estrutura de programação ou mesmo com recursos da linguagem, não receie, volte e leia o capítulo 2.

Capítulo 2
PHP para quem tem pressa

Não tenhamos pressa, mas não percamos tempo. – José Saramago

Neste capítulo faremos um curso de revisão (ou introdução) de PHP que abordará a construção de um cadastro de alunos. Esse cadastro será refeito diversas vezes, mostrando as várias formas de programar em PHP. Essa sequência de refatorações - este é o termo para dizer que as mudanças são evolutivas - nos lembrará ou fornecerá o conhecimento necessário para acompanhar o desenvolvimento da principal aplicação principal do livro, que é um sistema com dois cadastros, implementados com Programação Orientada a Objetos em uma arquitetura de múltiplas camadas.

Considere que este capítulo é como uma câmara de descompressão de um submarino (ou de uma espaçonave). Ele pode evitar que uma entrada brusca no terceiro capítulo cause algum desconforto. Você também pode considerar que está fazendo um alongamento antes de iniciar uma corrida e a leitura deste capítulo evitará distensões musculares.

Nosso diretório de trabalho será o `htdocs` do XAMPP. Todas as referências de sistema de arquivos sobre os programas a serem criados neste capítulo são baseadas na suposição de que o `htdocs` é o diretório raiz de nossos pequenos projetos de revisão (ou de introdução).

2.1 Um cadastro usando o sistema de arquivos

Nesta seção, nosso principal objetivo será dominar a manipulação do sistema de arquivos por uma aplicação web. Essa aplicação simulará uma parte de um sistema de administração de uma escola, com um cadastro de alunos e um cadastro de professores. Começaremos pelo de alunos.

O nosso cadastro terá uma página inicial que apresentará a lista de alunos cadastrados e terá um hyperlink para uma página que permitirá a inclusão de novos alunos. Além disso, essa página inicial permitirá a exclusão de alunos.

Esse será o projeto **escola1**. Crie o diretório `escola1` dentro do diretório `htdocs` . Dentro do diretório `escola1` , crie o arquivo `index.php` , que será a página inicial. Esse arquivo conterá um documento HTML com um hyperlink para uma página de formulário (`form.php`) e a importação de um outro arquivo (`listar.php`) que produzirá a listagem dos alunos.

Como nosso cadastro é um exemplo didático, temos apenas dois atributos do aluno, a matrícula e o nome. Se você souber manipular dois atributos, saberá manipular uma dúzia ou mais.

Estabelecido qual será o conteúdo do arquivo `index.php` , vamos preenchê-lo com o código a seguir.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Cadastro de Alunos</title>
</head>
<body>
    <h1>Cadastro de Alunos</h1>
    <a href="form.php">Incluir aluno</a>
    <table>
        <thead>
            <tr>
```

```
                <th>Matrícula</th>
                <th>Nome</th>
            </tr>
        </thead>
        <tbody>
<?php require 'listar.php'?>
        </tbody>
    </table>
</body>
</html>
```

O único trecho da linguagem PHP no arquivo `index.php` é o que importa o arquivo `listar.php` com o comando `require`. O comando `require` verifica se um arquivo existe antes de importá-lo, gera um erro fatal se ele não existir e interrompe o processamento do programa. É diferente do comando `include`, que tenta incluir e, se o arquivo não existir, apenas gera uma advertência, sem interromper o processamento do programa.

Como o arquivo `listar.php` não existe ainda, haverá um erro no processamento do PHP, que interromperá a renderização do arquivo `index.php` no trecho que compreende o conteúdo da tabela HTML. Mas já é possível ter uma ideia de como será a exibição dos dados, pois a parte estática visível não será afetada pelo erro.

No terminal, dentro do diretório `htdocs` inicie o servidor PHP embutido apontando para o diretório `escola1` como raiz. A instrução, tendo como base o XAMPP para Linux, fica assim:

```
../bin/php -S localhost:8000 -t escola1
```

O resultado no navegador web, ao acessar o endereço `localhost:8000`, será o da figura a seguir:

Figura 2.1: Página inicial do cadastro de alunos

Embora a página seja renderizada, se você observar a saída do servidor web no terminal, verá que há uma mensagem erro, conforme listagem a seguir, informando a ausência do arquivo `listar.php`. Essa mensagem também pode estar sendo exibida no cabeçalho da própria página se a diretiva `display_errors` do arquivo `php.ini` estiver configurada como `On` ou `1`.

```
127.0.0.1:43156 [500]: / - require(): Failed opening required 'listar.php'
(include_path='.:/usr/share/php') in /opt/lampp/htdocs/escola1/index.php
on line 18
```

Antes de criar o arquivo `listar.php`, criaremos o arquivo `form.php`, porque precisamos incluir os alunos primeiro para ter o que listar. No diretório `escola1`, crie o arquivo `form.php`, com o conteúdo a seguir, que define um formulário HTML:

```
<?php require 'aluno.php'?>
<!DOCTYPE html>
<html>
```

```
<head>
<meta charset="UTF-8">
<title>Inclusão de alunos</title>
</head>
<body>
<form method="post" action="gravar.php">
Nome <input type="text" name="nome" value="<?=$nome?>"
autofocus="autofocus">
<input type="hidden" name="matricula" value="<?=$matricula?>">
<input type="submit" value="gravar">
</form>
</html>
```

Você pode ter se incomodado com o fato de o código-fonte do arquivo `form.php` ter uma importação de um arquivo `aluno.php` no início. Esse arquivo será usado mais adiante, quando implementarmos a alteração de alunos cadastrados. A ausência desse arquivo impedirá a renderização da página definida por `form.php`, porque o comando `require` antecede o código HTML que define o formulário. Por isso, crie o arquivo `aluno.php` em `alunos1`. Por enquanto ele ficará vazio, pois nosso foco agora é a inclusão de alunos. Com esse arquivo criado, se você clicar no link **Incluir aluno** da página inicial, verá o formulário da figura a seguir.

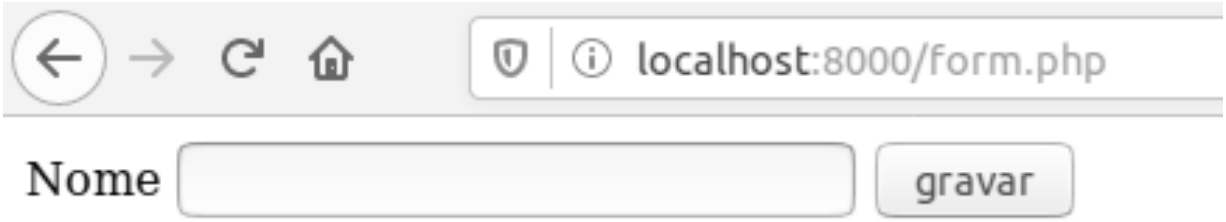


Figura 2.2: Formulário de inclusão de aluno

Conforme se percebe pelo atributo `action` do formulário, os dados são enviados para o arquivo `gravar.php` usando o método HTTP `POST` (definido no atributo `method`). Esse é o arquivo que fará a persistência dos dados em um arquivo de texto, de forma sequencial.

O arquivo `gravar.php` deve fazer as seguintes operações:

- Recuperar os dados de matrícula e nome da requisição HTTP POST usando a variável superglobal `$_POST` do PHP;
- Verificar se o nome do aluno foi enviado - se não foi, nada será feito e a página inicial será exibida;
- Definir o caminho absoluto para o arquivo de texto que armazenará os dados dos alunos (`alunos.txt`);
- Criar o arquivo `alunos.txt` se ele não existir e garantir que ele tem permissão para ser gravado;
- Abrir o arquivo `alunos.txt` em modo de leitura;
- Verificar se a requisição é para inclusão de um aluno (matrícula não informada) ou para alteração de um aluno (matrícula informada);
- Caso seja uma inclusão, lê o valor da última matrícula para gerar o próximo, fecha o arquivo e abre-o novamente em modo de gravação e adiciona o novo aluno ao final do arquivo;
- Caso seja uma alteração, cria um arquivo temporário e copia todos as linhas do arquivo `alunos.txt` para esse arquivo, modificando apenas a linha referente ao aluno alterado. Ao final do processo, substitui o arquivo `alunos.txt` pela cópia modificada.

Cada linha do arquivo `alunos.txt` corresponderá a um aluno. Esse arquivo terá linhas de comprimento fixo (80 caracteres), com a matrícula e o nome separados por vírgula.

As operações que definimos para o arquivo `gravar.php` serão implementadas pelo código da listagem a seguir:

```php
<?php
define('DS', DIRECTORY_SEPARATOR);

$nome = ($_POST['nome'] ?? NULL);
$matricula = ($_POST['matricula'] ?? NULL);

if (! is_null($nome)) {
    $filename = __DIR__ . DS . 'alunos.txt';

    require 'arquivo.php';

    // abre no começo para leitura
    $handle = fopen($filename, 'r');

    // inclusão
    if (empty($matricula)) {

        $ultimaMatricula = 0;
        while (! feof($handle)) {
            $record = explode(',', fread($handle, 80));
            $ultimaMatricula = (isset($record[0]) && empty($record[0])) ?
$ultimaMatricula : $record[0];
        }
        fclose($handle);
        $handle = fopen(__DIR__ . DS . 'alunos.txt', 'a');
        $matricula = $ultimaMatricula + 1;
        fwrite($handle, substr($matricula . ',' . $nome . ',' .
str_repeat(' ', 80), 0, 78) . ",\n", 80);
        fclose($handle);
    } else     // alteração
    {
        $tmpFilename = __DIR__ . DS . 'alunos.tmp';
        $tmpHandle = fopen($tmpFilename, 'w');
        while (! feof($handle)) {
            $row = fread($handle, 80);
            $record = explode(',', $row);
            $ultimaMatricula = (isset($record[0]) && empty($record[0])) ?
$ultimaMatricula : $record[0];
            if ($ultimaMatricula == $matricula) {
```

```
                fwrite($tmpHandle, substr($matricula . ',' . $nome . ',' .
str_repeat(' ', 80), 0, 78) . ",\n", 80);
            }
            else {
                fwrite($tmpHandle, $row);
            }
        }
        fclose($tmpHandle);
        fclose($handle);
        unlink($filename);
        copy($tmpFilename, $filename);
    }
}
header('Location:index.php');
```

Observe que na listagem anterior utilizamos:

- A função `define()` para criar uma constante de nome curto para o caractere separador de diretórios, que já é definido pela constante `DIRECTORY_SEPARATOR`. Ao usar a constante, garantimos que o código funcionará em Windows, Linux e MacOS;
- O operador de coalescência (??) para verificar se o array associativo `$_POST` possui uma determinada chave, evitando a leitura de um dado inexistente. O operador de coalescência retorna o valor do operando testado se ele existir e não for nulo. Caso contrário, retorna o segundo operando. O operador de coalescência é uma alternativa mais concisa que o operador ternário;
- A palavra reservada **DIR** que retorna o caminho absoluto para o arquivo PHP onde ela é referida;
- A função `is_null()` para verificar se a variável `$nome` é do tipo `null`;
- A função `empty()` para verificar se o nome não é um texto vazio;
- A função `fopen()` para abrir o arquivo `alunos.txt` e o arquivo temporário. O primeiro argumento dessa função é o nome do arquivo e o segundo é o modo de abertura. O modo `r` abre o arquivo no início, para leitura. O modo `w` abre o arquivo no final, para gravação;

- A função `fread()` lê uma quantidade determinada de bytes de um arquivo. Por isso é necessário definir um comprimento para os registros, para saber o quanto leremos a cada vez;
- A função `fwrite()` grava uma quantidade determinada de bytes em um arquivo. Como definimos o comprimento de 80 caracteres para nossos registros, garantimos que eles tenham esse tamanho, preenchendo com espaços em branco o que faltar;
- A função `feof()` indica se foi alcançado o final do arquivo e, portanto, se não há mais dados para serem lidos;
- A função `fclose()` fecha o arquivo identificado pela variável retornada pela função `fopen()`. Essa variável é do tipo `resource`, que não armazena um valor, mas um apontamento para um recurso do sistema operacional;
- A função `str_repeat()` para gerar uma sequência de 80 espaços em branco;
- A função `substr()` para extrair os 80 caracteres iniciais da concatenação da matrícula com o nome e com os 80 espaços em branco;
- A função `explode()` para transformar a última linha do arquivo em um array associativo com dois elementos, sendo o primeiro a matrícula, que precisamos ler para gerar a próxima;
- A função `unlink()` apaga o arquivo especificado como argumento. **Apaga mesmo**, não envia para lixeira;
- A função `copy()` copia um arquivo de um diretório para outro, com a possibilidade de mudar o nome no destino;
- A função `header()` define um cabeçalho para uma resposta HTTP. Neste caso estamos usando o argumento `location` para definir uma resposta HTTP 302, que faz um redirecionamento de página.

É claro que você reparou em duas coisas. A primeira é que o arquivo `gravar.php` importa um arquivo chamado `arquivo.php`. A segunda é que `gravar.php` não verifica se o arquivo `alunos.txt` existe antes de tentar abri-lo e nem se ele tem permissão de escrita. Essas duas coisas estão relacionadas. O código de verificação da existência do

arquivo e de sua permissão ficará no `arquivo.php`. Um bom motivo para deixar essa parte separada é que ela terá de ser executada na listagem dos alunos também. Crie esse arquivo com o código da listagem a seguir:

```
<?php
//Cria o arquivo se não existir
if (! file_exists($filename))
{
    if (!is_writable(__DIR__))
    {
        echo 'Não pode criar o arquivo alunos.txt';
        goto EOF;
    }
    $handle = fopen($filename, 'a');
    fclose($handle);
    chmod($filename, 0777);
}
EOF:
```

Na listagem anterior, se o arquivo não existe, usamos a função `fopen()` para criá-lo. O argumento `a` cria o arquivo se ele não existe antes de tentar abri-lo. A função `chmod()` altera as permissões do arquivo. Estamos usando 0777, que permite a leitura e gravação por todos os usuários. Isso é conveniente para desenvolvimento, mas procure restringir essas permissões para ambiente de produção. Os códigos para definição de permissões podem ser encontrados na própria documentação do comando `chmod` do Linux.

Com isso, implementamos a gravação do aluno em arquivo de texto. Mas da forma como está, ao final da gravação o programa apresentará a página inicial sem listar os alunos. Por isso, precisamos criar o arquivo `listar.php`. Esse arquivo lerá as linhas do arquivo `alunos.txt` e produzirá uma linha de tabela HTML para cada linha lida, com hyperlinks para alteração e exclusão. Crie o arquivo `listar.php` com o código da listagem a seguir:

```php
<?php
define('DS', DIRECTORY_SEPARATOR);

$filename = __DIR__ . DS . 'alunos.txt';

require 'arquivo.php';

$handle = fopen($filename, 'r');

while (! feof($handle)) {
    $record = explode(',', fread($handle, 80));
    if (! empty($record[0])) {
        echo '<tr>';
        echo "<td><a href=\"form.php?matricula={$record[0]}\">
    {$record[0]}</a></td>";
        echo "<td>{$record[1]}</td>";
        echo "<td><a href=\"apagar.php?matricula={$record[0]}\">
        Excluir</a></td>";
        echo '</tr>';
    }
}
fclose($handle);
```

Agora é possível não somente incluir alunos como ver os alunos incluídos (sem ter de abrir o arquivo `alunos.txt` diretamente). Se você, por exemplo, digitar **Huguinho** no formulário de inclusão e clicar no botão **gravar**, será redirecionado para a página inicial que exibirá o aluno incluído, conforme a imagem a seguir.

Figura 2.3: Página de listagem de alunos

Se continuar, incluindo o **Zezinho**, o **Luisinho** e a **Patrícia**, o resultado será este:

Figura 2.4: Página com vários alunos

O motivo de incluir mais alguns alunos, além de ter certeza de que a primeira inclusão não funcionou apenas por sorte, é para poder testar a alteração e a exclusão. Vamos começar com a exclusão. O arquivo `listar.php` gera a tabela HTML da lista de alunos. Essa tabela tem três colunas, sendo a última o hyperlink para excluir o aluno. Esse hyperlink chama o arquivo `apagar.php` passando o argumento `matricula` por HTTP `GET`.

Para excluir um registro de um arquivo sequencial, fazemos uma cópia de todos os registros, exceto daquele que se quer excluir, para um novo arquivo. Esse é o procedimento contido no arquivo `apagar.php`, que você deve criar com o conteúdo a seguir:

```
<?php
define('DS', DIRECTORY_SEPARATOR);

$matricula = ($_GET['matricula'] ?? NULL);

if (! is_null($matricula)) {
    $filename = __DIR__ . DS . 'alunos.txt';

    require 'arquivo.php';

    // leitura e abre no começo
    $handle = fopen($filename, 'r');

        $tmpFilename = __DIR__ . DS . 'alunos.tmp';
        $tmpHandle = fopen($tmpFilename, 'w');
        while (! feof($handle)) {
            $row = fread($handle, 80);
            $record = explode(',', $row);
            $ultimaMatricula = (isset($record[0]) && empty($record[0])) ?
$ultimaMatricula : $record[0];
            if ($ultimaMatricula != $matricula) {
                fwrite($tmpHandle, $row);
            }
        }
        fclose($tmpHandle);
        unlink($filename);
```

```
        copy($tmpFilename, $filename);
}
header('Location:index.php');
```

A única novidade do arquivo `apagar.php` em relação aos anterior é o uso da variável superglobal `$_GET`, que armazena dados enviados por HTTP `GET`, que é o caso da matrícula do aluno. Após criar o arquivo `apagar.php`, podemos excluir um aluno, como o Luisinho, por exemplo.

A última funcionalidade a ser implementada é a alteração do cadastro. Para habilitar essa funcionalidade precisamos preencher o arquivo `aluno.php` com um código que capture a matrícula do aluno, enviada por HTTP `GET` pelo hyperlink da página de listagem de alunos, e encontre a linha do arquivo que contém os dados do aluno. Esse código é apresentado na listagem a seguir:

```
<?php
define('DS', DIRECTORY_SEPARATOR);

$matricula = ($_GET['matricula'] ?? NULL);
$nome = '';

if (! is_null($matricula)) {
    $filename = __DIR__ . DS . 'alunos.txt';
    $handle = fopen($filename, 'r');

    while (! feof($handle)) {
        $record = explode(',', fread($handle, 80));
        if (! empty($record[0]) && $record[0] == $matricula) {
            $nome = $record[1];
        }
    }
    fclose($handle);
}
```

Com o arquivo `aluno.php` devidamente preenchido, podemos alterar os nomes dos alunos. E assim, finalizamos o nosso cadastro. A figura

a seguir ilustra o fluxo de dados entre os arquivos de nossa aplicação.

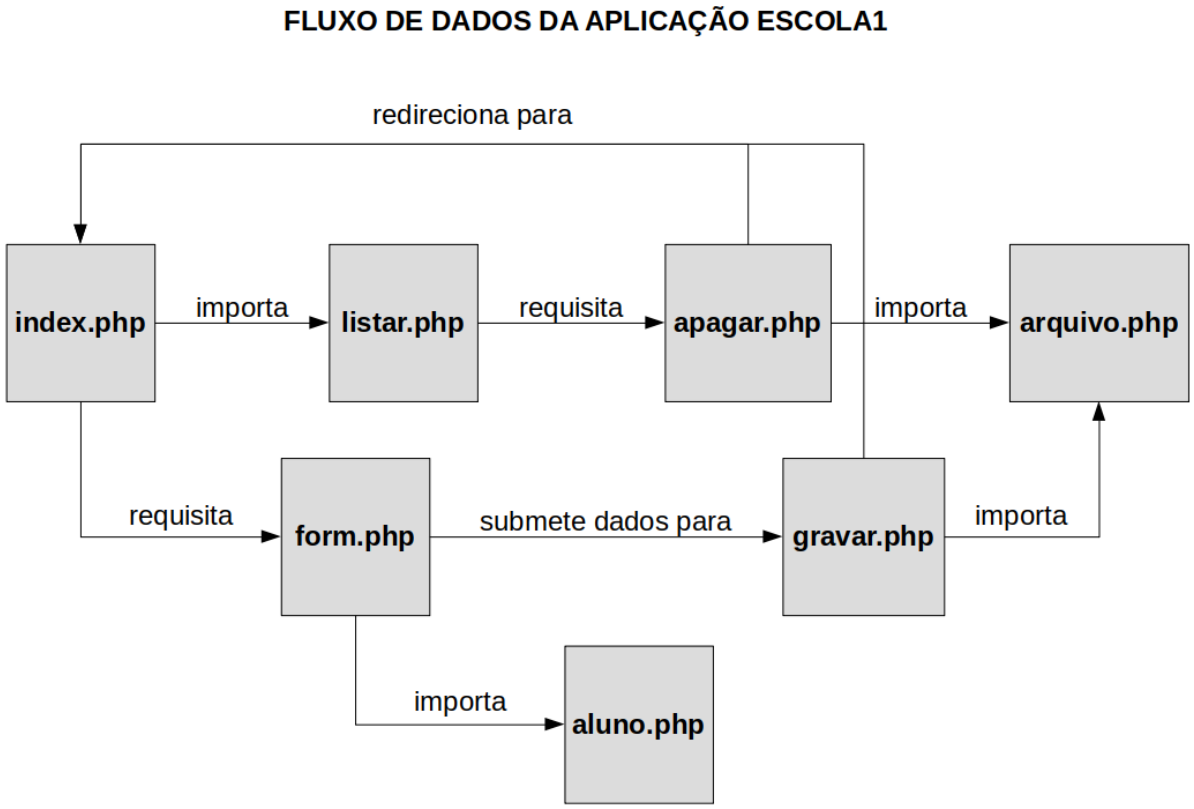


Figura 2.5: Fluxo de dados da aplicação alunos1

Na próxima seção, evoluiremos esse cadastro, substituindo o arquivo de texto por um banco de dados relacional.

2.2 Um cadastro usando banco de dados relacional

Já sabemos o que queremos fazer: incluir, alterar, excluir e listar alunos. A diferença é como faremos isso. O primeiro passo é copiar a pasta `escola1` como `escola2`.

Na pasta `escola2`, o arquivo `index.php`, que exibe a página inicial, permanecerá inalterado. O arquivo `form.php`, que exibe o formulário de inclusão ou alteração, também será exatamente o mesmo. As mudanças começarão pelo arquivo `gravar.php`. No lugar de ler e gravar um arquivo de texto, ele passará a executar comandos SQL. A lógica permanece a mesma. Se o arquivo `gravar.php` não receber um número de matrícula, ele executará um comando SQL `INSERT`, por meio do método `exec()` da classe `PDO`, que faz parte de uma extensão do PHP para bancos de dados. Se ele receber um número de matrícula, ele executará um comando SQL `UPDATE`. Sendo assim, o conteúdo do arquivo `gravar.php` de `escola2` será o da próxima listagem:

```
<?php
require 'pdo.php';

$nome = ($_POST['nome'] ?? NULL);
$matricula = (integer) ($_POST['matricula'] ?? NULL);

if (! is_null($nome)) {
    $sql = "INSERT INTO alunos(nome) values ('$nome')";
    if (!empty($matricula)) {
        $sql =
        "UPDATE alunos SET nome='$nome'
        WHERE matricula=$matricula";
    }

   if (! $pdo->exec($sql)) {
     echo 'Não conseguiu gravar o registro';
     exit();
   }
}
header('Location:index.php');
```

Observe que o arquivo `gravar.php` importa um arquivo `pdo.php`, que contém as instruções para conexão a um banco de dados com uma tabela de alunos. O banco e a tabela têm de existir. Por isso vamos criá-los antes de criar o arquivo `pdo.php`.

Você pode usar o `phpmyadmin` que vem com o XAMPP para criar o banco e a tabela. Para isso, é necessário iniciar não somente o servidor MySQL, conforme foi mencionado no primeiro capítulo, mas também o Apache. Com os dois serviços iniciados, você pode acessar o `phpmyadmin` por meio do endereço http://localhost/phpmyadmin/. Por padrão, no XAMPP ele vem instalado com um usuário administrador `root` sem senha.

Após acessar o `phpmyadmin`, clique na aba `Bancos de dados` no quadro da direita e será aberto um formulário para criação de um banco. Preencha-o com escola como nome de banco e `utf8_unicode_ci` (ou com outro conjunto UTF8 com sufixo `unicode_ci`) como o padrão de codificação de caracteres.

Após clicar no botão `Criar`, a página será atualizada e o banco `escola` aparecerá, tanto na estrutura em árvore no quadro da esquerda como na lista do quadro da direita. Clique no nome do banco de dados para selecioná-lo e, em seguida, na aba SQL no quadro da direita. O script com os comandos SQL de criação do banco de dados está disponível no GitHub. Copie para o quadro `Rodar consulta` o conteúdo de https://github.com/fgsl/phpbooksample/tree/main/sql/alunos.sql. Esse arquivo contém o comando `SQL CREATE TABLE` para uma tabela de alunos. Após clicar no botão `Executar`, a tabela `alunos` será criada.

Agora, com banco e tabela criados, podemos implementar o arquivo `pdo.php`. O arquivo tem esse nome porque contém a criação de um objeto da classe `PDO` que faz a conexão com o banco de dados `escola`. Veja o conteúdo do arquivo:

```
<?php
$pdo = new PDO('mysql:dbname=escola;host=localhost','root', '');
```

A classe `PDO` (de PHP Data Objects) implementa uma camada de abstração para bancos de dados em PHP, fornecendo atributos e métodos únicos para diferentes marcas de sistemas gerenciadores

de bancos de dados. O primeiro argumento do construtor da classe é a identificação do banco de dados (a marca, o nome e o servidor), o segundo, o nome de usuário e o terceiro é a senha.

O arquivo `pdo.php`, com apenas uma instrução, vai substituir o `arquivo.php`, que pode ser removido de `escola2`. O trabalho de abertura, fechamento, leitura e gravação de arquivo de texto está sendo substituído pelo acesso a um sistema gerenciador de banco de dados relacional, que passará a gravar e ler os dados de nossa aplicação. Se o `arquivo.php` foi substituído pelo `pdo.php`, todas as importações do primeiro têm de ser substituídas pelo segundo. Além disso, as referências ao arquivo de texto têm de ser substituídas pela referência ao objeto da classe `PDO`.

Vamos alterar o arquivo `listar.php` para que ele gere a tabela HTML de alunos a partir de uma consulta à tabela SQL de alunos. Para recuperar registros de uma tabela com a classe `PDO`, usamos o método `query()`, que recebe como argumento um comando SQL `SELECT`. O método `query()` retorna um objeto da classe `PDOStatement`, a partir do qual podemos recuperar os registros usando o método `fetchAll()`. Os registros são recuperados como um array de arrays associativos onde os campos da tabela são as chaves. Com a substituição das funções de manipulação de arquivos pelo objeto da classe `PDO`, o arquivo `listar.php` terá como conteúdo a listagem a seguir:

```
<?php
require 'pdo.php';

$resultSet = $pdo->query('SELECT * FROM alunos');

$records = $resultSet->fetchAll();
foreach($records as $record)
{
    echo '<tr>';
    echo
```

```
    "<td><a href=\"form.php?matricula={$record['matricula']}\">
    {$record['matricula']}</a></td>";
    echo "<td>{$record['nome']}</td>";
    echo
    "<td><a href=\"apagar.php?matricula={$record['matricula']}\">
    Excluir</a></td>";
    echo '</tr>';
}
```

O arquivo `apagar.php` também será alterado, para apagar o registro de um aluno usando o comando SQL `SELECT`. O conteúdo modificado desse arquivo é exibido a seguir:

```
<?php
require 'pdo.php';

$matricula = (integer) (isset($_GET['matricula']) ? $_GET['matricula']
: NULL);

$pdo->exec("DELETE FROM alunos WHERE matricula=$matricula");

header('Location: index.php');
```

Finalmente, vamos modificar o arquivo `aluno.php`, que recupera o registro de um aluno para alteração. Usaremos o método `query()` da classe `PDO` para fazer um SQL `SELECT` com filtro para a matrícula que foi enviada por HTTP `GET`. Mas no lugar de usar o método `fetchAll()` de `PDOStatement` para recuperar o registro, usamos o método `fetchColumn()` porque precisamos recuperar apenas um campo (ou coluna), que é o nome, pois já temos a matrícula. O método `fetchColumn()` devolve o n-ésimo campo do registro, sendo *n* um número inteiro passado como argumento.

```
<?php
require 'pdo.php';
define('DS', DIRECTORY_SEPARATOR);

$matricula = ($_GET['matricula'] ?? NULL);
$nome = '';
```

```
if (! is_null($matricula)) {

    $resultSet = $pdo->query("SELECT * FROM alunos WHERE
matricula=$matricula");
    $nome = $resultSet->fetchColumn(1);
}
```

Você pode interromper a execução da aplicação `escola1` usando a combinação de teclas `CTRL+C`, e iniciar a aplicação `escola2`. Conforme se vê na instrução a seguir, o que mudará de agora em diante na execução do servidor embutido é apenas a indicação do diretório web com o argumento `-t`:

```
../bin/php -S localhost:8000 -t escola2
```

A figura a seguir ilustra o fluxo de dados entre os arquivos da aplicação `escola2`. Comparando com a aplicação `escola1`, o único arquivo diferente é o `pdo.php`, que tomou o lugar de `arquivo.php`. É possível verificar que a delegação da manipulação dos dados para o banco de dados relacional nos proporcionou uma pequena redução de linhas de código.

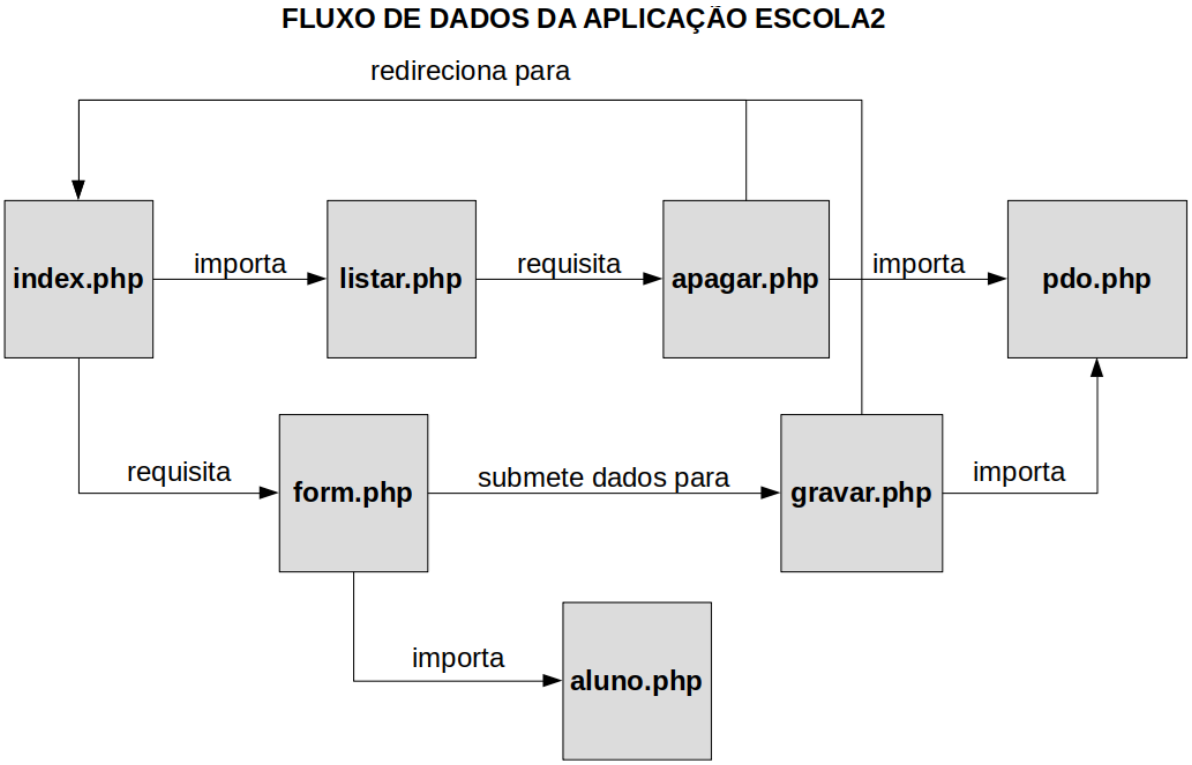


Figura 2.6: Fluxo de dados da aplicação escola2

Conforme visto nesta seção, usamos uma classe, mas nossa aplicação não está orientada a objetos. É importante comentar isso pois o fato de haver classes em uma aplicação de software não implica que sua estrutura seja orientada a objetos.

A classe é uma estrutura que permite o reaproveitamento de rotinas que se repetem em um programa de computador. Mas antes de explorar mais a estrutura de classes, vamos falar de outra estrutura de reúso que a antecede historicamente em programação, que é a função. Na próxima seção, reescreveremos nossa aplicação usando funções.

2.3 Um cadastro com função definida pelo programador

Nas aplicações `escola1` e `escola2` usamos diversas funções da linguagem PHP. Funções são subrotinas de um programa, que evitam a repetição de um bloco de instruções pela atribuição de um identificador a ele. Existem centenas de funções previamente definidas nas diversas extensões da linguagem PHP, mas você também pode criar as suas. Nesta seção, mostraremos como criar um novo cadastro em nossa aplicação, reusando código com funções.

O primeiro passo é copiar a pasta `escola2` como `escola3`. Na pasta `escola3`, vamos alterar o arquivo `index.php`. Ele deixará de ser a página inicial do cadastro de alunos para ser a página inicial do sistema da escola, com hyperlinks para os cadastros de alunos e professores. Por isso seu conteúdo deve ser alterado para o que mostra a listagem a seguir:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Cadastros</title>
</head>
<body>
<h1>Cadastros</h1>
<a href="alunos.php?cadastro=alunos">Alunos</a><br>
<a href="professores.php?cadastro=professores">Professores</a>
</body>
</html>
```

O arquivo `alunos.php` não existe. Ele terá como conteúdo a página inicial do cadastro de alunos (que era o conteúdo anterior do arquivo `index.php`). Mas haverá duas modificações em relação à aplicação `escola2`: o hyperlink para o formulário de inclusão passará a enviar um parâmetro com a identificação do cadastro e será criada uma variável para identificar o cadastro, para uso do arquivo `listar.php`. O conteúdo de `alunos.php` será o seguinte:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Cadastro de Alunos</title>
</head>
<body>
    <h1>Cadastro de Alunos</h1>
    <a href="form.php?cadastro=alunos">Incluir aluno</a>
    <table>
        <thead>
            <tr>
                <th>Matrícula</th>
                <th>Nome</th>
            </tr>
        </thead>
        <tbody>
<?php
$cadastro = 'alunos';
require 'listar.php'
?>
</tbody>
    </table>
    <a href="index.php">Homepage</a>
</body>
</html>
```

O arquivo `listar.php` terá que ser modificado, para capturar o nome do cadastro e repassá-lo para os links de alteração e exclusão, conforme vemos na listagem a seguir:

```
<?php
require 'pdo.php';

$cadastro = isset($_GET['cadastro']) ? $_GET['cadastro'] : NULL;

$resultSet = $pdo->query("SELECT * FROM $cadastro");

$records = $resultSet->fetchAll();
```

```
foreach($records as $record)
{
    $chave = array_key_first($record);
    echo '<tr>';
    echo
    "<td><a href=\"form.php?$chave={$record[$chave]}&cadastro=$cadastro\">
    {$record[$chave]}</a></td>";
    echo "<td>{$record['nome']}</td>";
    echo
    "<td><a href=\"apagar.php?$chave=
{$record[$chave]}&cadastro=$cadastro\">
    Excluir</a></td>";
    echo '</tr>';
}
```

Observe que usamos a função `array_key_first` para obter o nome do primeiro campo da tabela, que é a chave primária.

Conforme avisamos, o arquivo `form.php` passa a enviar por HTTP `GET` um parâmetro chamado `cadastro`, com o valor **alunos**. Isso indica que o arquivo `form.php` será modificado, pois fará uso desse parâmetro. Além disso, ele repassará esse parâmetro por HTTP `POST` para o arquivo `gravar.php` em um campo HTML do tipo `hidden`. O arquivo `form.php` ficará como mostra a próxima listagem:

```
<?php require 'entidade.php'?>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Inclusão de <?=$cadastro?></title>
</head>
<body>
<form method="post" action="gravar.php">
Nome <input type="text" name="nome" value="<?=$nome?>"
autofocus="autofocus">
<input type="hidden" name="chave" value="<?=$chave?>">
<input type="hidden" name="cadastro" value="<?=$cadastro?>">
<input type="submit" value="gravar">
```

```
</form>
</html>
```

Você pode ter ficado decepcionado, pois não encontrou em `form.php` uma referência a `$_GET['cadastro']`, que deveria aparecer para a leitura do parâmetro enviado pelo arquivo `index.php`. E pode ter ficado intrigado ao ver duas variáveis (`$cadastro` e `$chave`) que são usadas mas não são criadas nesse arquivo. A explicação para a ausência de `$_GET['cadastro']` e para a presença de `$cadastro` e `$chave` está em outra alteração. Na aplicação `escola2`, o arquivo `form.php` importava o arquivo `aluno.php`, que implementava uma consulta ao registro do aluno. Como esse formulário será usado tanto para o cadastro de alunos como para o de professores, a importação do arquivo `aluno.php` foi substituída pelo arquivo `entidade.php`, que pode consultar tanto alunos quanto professores. Apague o arquivo `aluno.php` e crie o `entidade.php`, com o seguinte conteúdo:

```
<?php
require 'pdo.php';
require 'functions.php';

$cadastro = isset($_GET['cadastro']) ? $_GET['cadastro'] : NULL;
$nomeChave = getChave($cadastro);

$chave = isset($_GET[$nomeChave]) ? $_GET[$nomeChave]
: NULL;
$nome = '';

if (!is_null($chave)){
    $resultSet = $pdo->query(
        "SELECT * FROM $cadastro WHERE $nomeChave=$chave");

    $nome = $resultSet->fetchColumn(1);
}
```

O arquivo `entidade.php` constrói um comando SQL `SELECT` usando três variáveis: `$cadastro`, `$nomeChave` e `$chave`. A primeira e a terceira

são obtidas pela variável superglobal `$_GET`. A segunda é obtida com a função `getChave()`. Essa não é uma função da linguagem de programação PHP, mas uma função que nós criamos para este projeto. Essa função é definida no arquivo `functions.php`. Crie esse arquivo com o conteúdo a seguir:

```php
<?php
/**
 * Retorna o nome da chave de um cadastro
 * @return string|NULL
 */
function getChave(?string $cadastro = null)
{
    switch ($cadastro) {
        case 'alunos':
           return 'matricula';
        case 'professores':
           return 'codigo';
    }
    return NULL;
}
```

Uma função recebe um ou mais argumentos de entrada, que são manipulados como se fossem variáveis locais. E retorna apenas um tipo de dados como saída, embora esse tipo possa ser uma coleção de dados, como um `array`. Uma função pode retornar um valor sem receber argumentos explicitamente, como em `$resultado = funcaoQueFazAlgumaCoisa();`. Neste caso, ela pode ser uma função constante, que sempre retorna o mesmo valor, ou utilizar dados que não são fornecidos por quem a chamou para produzir a saída.

Uma função que retorna a hora atual, por exemplo, pode dispensar um argumento de entrada, pois vai utilizar a informação do sistema operacional para determinar sua saída. Se uma função PHP não retorna um valor, e portanto não pode ser atribuída a uma variável, ela na verdade é um procedimento. Algumas linguagens de programação têm sintaxes diferentes para funções e procedimentos, mas em PHP ambas usam a construção `function`.

Conforme vimos no arquivo `form.php`, o arquivo `gravar.php` recebe um parâmetro `cadastro`. Esse parâmetro é usado para determinar em qual tabela os dados serão gravados. Além disso, ele é usado para redirecionar o usuário para a página inicial do cadastro que foi alterado. Havendo compreendido quais serão as mudanças, altere o arquivo `gravar.php` para que seu conteúdo corresponda à listagem a seguir:

```
<?php
require 'pdo.php';
require 'functions.php';

$cadastro = isset($_POST['cadastro']) ? $_POST['cadastro'] : NULL;

if (is_null($cadastro)){
        echo 'Cadastro não informado';
        exit();
}

$nomeChave = getChave($cadastro);

$nome = isset($_POST['nome']) ? $_POST['nome'] : NULL;
$chave = (isset($_POST['chave']) ? (integer)$_POST['chave'] : NULL);

if (! is_null($nome)) {
    $sql = "INSERT INTO $cadastro(nome) values ('$nome')";

    if (!empty($chave) ) {
        $sql =
        "UPDATE $cadastro SET nome='$nome'
        WHERE $nomeChave=$chave";
    }
    if (! $pdo->exec($sql)) {
        echo 'Não conseguiu gravar o registro';
        exit();
    }
}

header("Location:$cadastro.php?cadastro=$cadastro");
```

O procedimento de exclusão de um registro também será parametrizado, para que o arquivo `apagar.php` seja capaz de remover registros tanto da tabela de alunos quanto da tabela de professores. Altere o conteúdo do arquivo `apagar.php` para que reflita a listagem a seguir:

```
<?php
require 'pdo.php';
require 'functions.php';

$cadastro = isset($_GET['cadastro']) ? $_GET['cadastro'] : NULL;
$nomeChave = getChave($cadastro);

$chave = (integer) (isset($_GET[$nomeChave]) ? $_GET[$nomeChave] : NULL);

$pdo->exec("DELETE FROM $cadastro WHERE $nomeChave=$chave");

header("Location: $cadastro.php?cadastro=$cadastro");
```

A partir do momento em que padronizamos a estrutura de inclusão, alteração e exclusão de registros, a criação do cadastro de professores se torna muito mais fácil, pois reaproveitaremos a maior parte da implementação. Na verdade, basta criarmos um arquivo, chamado `professores.php` , que será a página inicial do cadastro de professores, conforme a listagem a seguir:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Cadastro de Professores</title>
</head>
<body>
    <h1>Cadastro de Professores</h1>
    <a href="form.php?cadastro=professores">Incluir professor</a>
    <table>
        <thead>
            <tr>
                <th>Código</th>
```

```
                <th>Nome</th>
            </tr>
        </thead>
        <tbody>
<?php
$cadastro='professores';
require 'listar.php'
?>
</tbody>
    </table>
    <a href="index.php">Homepage</a>
</body>
</html>
```

Observe que esse arquivo é parecido em estrutura com o `alunos.php`. O que muda é a identificação do cadastro. Para adicionar mais cadastros, com a estrutura que criamos, teremos a criação de um arquivo a cada novo cadastro. Sem essa estrutura de reaproveitamento, teríamos pelo menos 7 novos arquivos a cada cadastro adicionado e muito código ficaria repetido.

Pensar em reaproveitamento no início de um projeto de software evita que a manutenção se torne mais complicada, pela quantidade de arquivos que compõem a aplicação.

Na figura a seguir o fluxo de dados mostra como reaproveitamos código, ao mostrar como um mesmo caminho serve para mais de um cadastro.

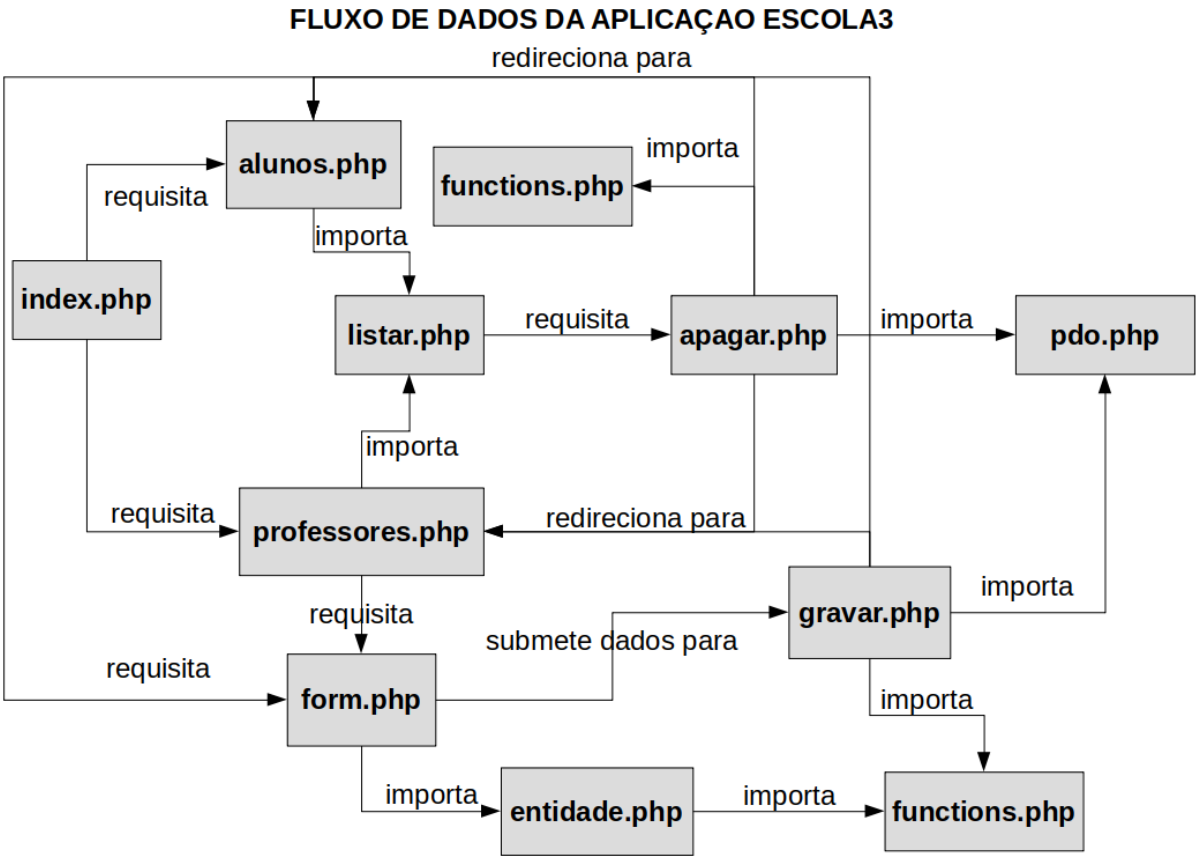


Figura 2.7: Fluxo de dados da aplicação escola3

Na próxima seção, vamos usar Programação Orientada a Objetos para reduzir a quantidade de código PHP nos arquivos do sistema escolar.

Mas, você pode perguntar, e o cadastro de professores? O cadastro de professores só funcionará se você criar a tabela de professores. O script de criação dessa tabela está disponível em https://github.com/fgsl/phpbooksample/tree/main/sql/professores.sql.

2.4 Um cadastro com uma classe abstrata e duas classes concretas

Nós usamos uma classe a partir da segunda versão do nosso sistema escolar. Agora vamos criar nossas próprias classes. Classes são estruturas que combinam dados e funções, chamadas nesse contexto de atributos e métodos, respectivamente. O principal propósito de uma classe é evitar a repetição de código-fonte ao longo de um projeto de software. A classe é um passo adiante da função no reaproveitamento de código-fonte.

Copie a pasta `escola3` como `escola4`. O arquivo `index.php` permanecerá inalterado. As mudanças começarão pelos arquivos `alunos.php` e `professores.php`. Ambos esses arquivos passarão a utilizar classes para manipular os registros de suas respectivas tabelas. Antes de alterá-los, criaremos as classes que eles utilizarão.

Anteriormente, você percebeu que há similaridades entre os dois cadastros. O fato de haver dados e funções em comum permite pensar em uma generalização dos dois cadastros. Essa generalização é expressa por uma classe abstrata, que contém os dados e operações comuns.

Vamos deixar essa classe separada na estrutura da aplicação, em uma pasta chamada `Entidade`. Dentro dessa pasta, crie o arquivo `EntidadeAbstrata.php`. Ele conterá uma classe chamada `EntidadeAbstrata`. Vamos criar passo a passo seu conteúdo. Primeiro, crie a estrutura da classe, de acordo com o código-fonte a seguir:

```
<?php
namespace Entidade;

abstract class EntidadeAbstrata
{
}
```

A palavra-chave `namespace` identifica um espaço dentro do qual um conjunto de nomes de classe existe. Isso permite a existência de classes com o mesmo nome, desde que identificadas com *namespaces* diferentes. A palavra-chave `abstract` informa que essa classe não pode ser usada diretamente para gerar uma instância, ou em outras palavras, para criar um objeto. É diferente da classe `PDO` que usamos anteriormente para estabelecer uma conexão com o banco de dados. A classe `PDO` é uma classe concreta, por isso é possível gerar um objeto a partir dela com o operador `new`. Se tentarmos criar um objeto da classe `EntidadeAbstrata` com o operador `new` teremos como resultado um erro fatal. O objetivo da classe abstrata é manter e distribuir um código que seria repetido por várias classes. Ela é uma fonte de cópia, essencialmente.

Agora vamos criar os atributos dessa classe. Há dois dados que são necessários tanto para o cadastro de alunos quanto para o cadastro de professores: a conexão com o banco de dados e o nome da tabela que será manipulada. Esses dois dados serão armazenados em atributos estáticos, que são atributos da classe e não precisam de um objeto para serem lidos ou gravados. Implemente os atributos `$pdo` e `$tabela` conforme a listagem:

```
    protected static ?\PDO $pdo = NULL;

    protected static $tabela = NULL;
```

Atributos em classes têm visibilidade controlada por identificadores em sua declaração. O identificador `public` permite a livre leitura e gravação de um atributo por qualquer outro objeto. O `private` impede a leitura e gravação de um atributo por outro objeto. O `protected` restringe a leitura e gravação a classes descendentes. É o caso dos atributos da classe `EntidadeAbstrata`, que serão herdados pelas classes concretas que criaremos adiante.

Mas antes de criá-las, precisamos implementar os métodos da classe `EntidadeAbstrata`. O primeiro a ser implementado é o `__call()`, que é um método mágico do PHP, assim como o método `__construct()`.

Métodos mágicos não são chamados pelo programador ou por outros objetos. Eles são invocados a partir de determinados eventos. O `__construct()`, por exemplo, é invocado quando o operador `new` é executado. O método `__call()` é chamado quando um objeto recebe uma chamada a um método que ele não define, ou seja, a um método inexistente. Ele permite interceptar uma chamada que resultaria em erro e produzir um resultado válido.

Neste caso, produziremos um conjunto virtual e genérico de métodos leitores e gravadores de atributos. Isso quer dizer que, se uma classe herdeira de `EntidadeAbstrata` tiver um atributo `$nome`, privado ou protegido, ele poderá ser lido com um método virtual `getNome()` e gravado com um método virtual `setNome()`. Chamamos de virtuais porque esses métodos não serão declarados, isto é, não existem realmente na classe. Mas eles funcionarão, como se existissem, graças à implementação da listagem a seguir, que você deve incluir na classe `EntidadeAbstrata`.

```
/**
 * get/set genéricos
 * @param string $method
 * @param array $args
 * @return mixed
 */
public function __call($method, array $args)
{
    $prefix = substr($method,0,3);
    if ($prefix == 'get')
    {
        $attribute = lcfirst(substr($method,3));
        return $this->$attribute;
    }
    if ($prefix == 'set')
    {
        $attribute = lcfirst(substr($method,3));
        $this->$attribute = $args[0];
        return $this;
```

```
        }
    }
```

O método `__call()` recebe como argumentos o nome do método chamado e o conjunto de parâmetros que foi passado para ele. A partir do nome do método chamado, chegamos ao nome do atributo. E usando a funcionalidade de variáveis variáveis do PHP conseguimos manipular dinamicamente qualquer atributo. Uma variável variável é uma variável cujo nome é referenciado por outra variável. Na implementação do método `__call()` temos a expressão `$this->$attribute`. Não existe o atributo `$this->$attribute`, nem na classe `EntidadeAbstrata` e nem nas classes herdeiras, que ainda serão definidas.

Cada cadastro manipula uma tabela, que possui uma chave primária. Nos projetos anteriores, a informação da chave se mostrou essencial para recuperar, alterar ou apagar registros. Por isso, a classe `EntidadeAbstrata` definirá um método para recuperar o nome da chave do cadastro que a classe herdeira vai manipular. A classe `EntidadeAbstrata` definirá apenas a assinatura desse método, sem o implementar, pois isso será feito por cada classe herdeira, uma vez que a implementação será diferente. Um método que tem apenas a assinatura (que é o conjunto de seu nome e dos identificadores) é um método abstrato. Para ser válido - o que significa ser compilado sem erro - um método abstrato precisa ser precedido pela palavra `abstract` e não deve ter corpo delimitado por chaves. Ele se encerra como uma instrução comum, com ponto e vírgula. É assim que você definirá o método `getChave()` conforme listagem a seguir:

```
    abstract public static function getChave();
```

O atributo `$pdo` que declaramos no início da classe será populado pelo método `getPdo()`, que encapsulará os dados de conexão com o banco de dados. O método `getPdo()` representa o encapsulamento do arquivo `pdo.php` pela classe `EntidadeAbstrata`. Implemente o método `getPdo()` de acordo com a listagem a seguir:

```
    public static function getPdo(): \PDO
    {
        if (self::$pdo == NULL)
        {
            self::$pdo = new
\PDO('mysql:dbname=escola;host=localhost','root', '');
        }
        return self::$pdo;
    }
```

A listagem dos registros ocorre de forma repetida no projeto `escola3`. Essa repetição será eliminada no `escola4` pela implementação genérica da listagem na classe `EntidadeAbstrata`. O método que fará isso será o `listar()`, que encapsulará o código existente nos arquivos `alunos.php` e `professores.php`. Esse método será estático, pois não será necessário armazenar dados, apenas lê-los. Ele é definido no trecho de código-fonte a seguir:

```
    public static function listar()
    {
        $resultSet = self::getPdo()->query('SELECT * FROM '.
static::$tabela);

        $records = $resultSet->fetchAll();

        $html = '';

        $chave = static::getChave();

        $cadastro = static::$tabela;

        foreach($records as $record)
        {
            $html .= <<<BLOCO
            <tr>
            <td>
            <a href="form.php?cadastro=$cadastro&chave={$record[$chave]}">
            {$record[$chave]}</a>
            </td>
```

```
            <td>{$record['nome']}</td>
            <td>
            <a href="apagar.php?cadastro=$cadastro&chave=
{$record[$chave]}">
            Excluir</a>
            </td>
            </tr>
BLOCO;
        }
        return $html;
    }
```

Se você comparar esse método `listar` com o código do arquivo `listar.php`, verá que o método é mais eficiente na determinação da chave da tabela para uso na listagem, pois recupera o nome apenas uma vez, enquanto o anterior repete a consulta para cada registro. Além disso, em vez de usar vários comandos `echo` para imprimir o texto HTML de forma fragmentada, usamos a sintaxe `heredoc` para definir o bloco de texto HTML de uma só vez.

Para alterar um registro, precisamos recuperar os dados do registro da tabela. As instâncias das classes herdeiras de `EntidadeAbstrata` servirão para encapsular um registro. Para isso, em vez de criar diretamente as instâncias com o operador `new`, vamos usar um método que fará uma consulta à tabela do cadastro e populará o objeto com o registro encontrado. Esse método receberá o valor da chave primária a ser localizado como argumento. Como ele saberá em qual tabela ele deverá fazer a consulta? A partir do valor do atributo `$tabela` da classe concreta. Entendido isso, implemente o método `get()` com a listagem a seguir:

```
    public static function get($chave): object
    {
        $nomeChave = static::getChave();

        $nome = '';

        $cadastro = static::$tabela;
```

```
        if (!is_null($chave)){
            $resultSet = self::getPdo()->query(
                "SELECT * FROM $cadastro WHERE $nomeChave=$chave");

            $nome = $resultSet->fetchColumn(1);
        }

        $class = get_called_class();
        $method = 'set' . ucfirst($nomeChave);

        $entidade = new $class();
        $entidade->$method($chave)
        ->setNome($nome);

        return $entidade;
    }
```

Outra função a ser generalizada é a gravação do registro, que pode ser uma inclusão ou atualização. Essa função no projeto `escola3` é feita pelo arquivo `gravar.php`. Esse arquivo continuará existindo no projeto `escola4`, mas transferirá uma parte de sua responsabilidade para o método `gravar()` da classe `EntidadeAbstrata`. Esse é um método de instância, é necessário ter um objeto da classe para utilizá-lo. Sua implementação, a qual você deve reproduzir, encontra-se logo a seguir:

```
    public function gravar(array $dados)
    {
        $nomeChave = static::getChave();

        $nome = isset($dados['nome']) ? $dados['nome'] : NULL;
        $chave = (integer) (isset($dados['chave']) ? $dados['chave'] :
NULL);

        $cadastro = static::$tabela;

        if (! is_null($nome)) {
            $sql = "INSERT INTO $cadastro(nome) values ('$nome')";
```

```
            if (!empty($chave)) {
                $sql =
                "UPDATE $cadastro SET nome='$nome'
                WHERE $nomeChave=$chave";
            }

            if (self::getPdo()->exec($sql) === false) {
                throw new \Exception('Não conseguiu gravar o registro');

            }
        }
    }
```

Finalmente, temos o método para apagar um registro, aquele método que sempre funciona, porque destruir é sempre mais fácil que construir. Esse método é o `apagar()`, que se apropriará de (grande) parte da responsabilidade do arquivo `apagar.php`. Implemente esse método de acordo com o código-fonte a seguir:

```
    public function apagar($chave)
    {
        $chave = (integer) $chave;

        $nomeChave = static::getChave();

        $cadastro = static::$tabela;

        self::getPdo()->
        exec("DELETE FROM $cadastro WHERE $nomeChave=$chave");

    }
}
```

Isolar as partes repetitivas de um software é uma tarefa trabalhosa, mas, uma vez feita, permite evitar longas digitações.

Agora que terminamos nossa classe abstrata, podemos criar as classes concretas. Elas serão bastante concisas, pois terão somente

o que é diferente uma da outra. Vamos criar primeiro, no arquivo `Aluno.php`, na pasta `Entidade`, a classe `Aluno` com o conteúdo a seguir:

```php
<?php
namespace Entidade;

class Aluno extends EntidadeAbstrata
{
    protected ?int $matricula;
    protected ?string $nome;

    protected static $tabela = 'alunos';

    public static function getChave()
    {
        return 'matricula';
    }
}
```

Em seguida, crie no arquivo `Professor.php`, ainda na pasta `Entidade`, a classe `Professor`, desta forma:

```php
<?php
namespace Entidade;

class Professor extends EntidadeAbstrata
{
    protected ?int $codigo;
    protected ?string $nome;

    protected static $tabela = 'professores';

    public static function getChave()
    {
        return 'codigo';
    }
}
```

Agora temos as duas classes concretas dos dois cadastros de nosso pequeno sistema de informação escolar. Cada uma declara os atributos específicos da entidade que manipula e implementa o método abstrato definido pela classe abstrata. Vamos agora revisar os arquivos que foram copiados do projeto `escola3`.

Vamos começar pelas páginas iniciais dos cadastros. Os arquivos `alunos.php` e `professores.php` importam o arquivo `listar.php` para produzir respectivamente a listagem de alunos e professores. Esse código foi encapsulado pela classe `EntidadeAbstrata`, por isso nós substituiremos a implementação da listagem de registros nesses arquivos pela chamada ao método `listar()` das classes herdeiras. Altere o arquivo `alunos.php` de modo que fique como o trecho de código-fonte apresentado a seguir:

```
<?php
require 'functions.php';
use Entidade\Aluno;
?>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Cadastro de Alunos</title>
</head>
<body>
    <h1>Cadastro de Alunos</h1>
    <a href="form.php?cadastro=alunos">Incluir aluno</a>
    <table>
        <thead>
            <tr>
                <th>Matrícula</th>
                <th>Nome</th>
            </tr>
        </thead>
        <tbody>
<?php
echo Aluno::listar();
```

```
?>
</tbody>
    </table>
    <a href="index.php">Homepage</a>
</body>
</html>
```

O uso do método `listar()` da classe `Aluno` tornou desnecessária a importação do arquivo `listar.php`. Você pode apagar esse arquivo sem medo, pois ele não será mais utilizado. Vamos alterar agora o arquivo `professores.php` para usar o método `listar()` da classe `Professor`. O código-fonte que você deverá ter no arquivo `professores.php` será o seguinte:

```
<?php
require 'functions.php';
use Entidade\Professor;
?>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Cadastro de Professores</title>
</head>
<body>
    <h1>Cadastro de Professores</h1>
    <a href="form.php?cadastro=professores">Incluir professor</a>
    <table>
        <thead>
            <tr>
                <th>Código</th>
                <th>Nome</th>
            </tr>
        </thead>
        <tbody>
<?php
echo Professor::listar();
?>
</tbody>
```

```
    </table>
    <a href="index.php">Homepage</a>
</body>
</html>
```

Você deve ter reparado que ambos os arquivos `alunos.php` e `professores.php` importam o arquivo `functions.php`. Esse arquivo já existia no projeto `escola3` contendo apenas a função `getChave()`. Essa função foi encapsulada pela classe `EntidadeAbstrata` e portanto não precisa mais existir no arquivo `functions.php`. Mas esse arquivo conterá agora três novas funções. A função `getCadastro()` recupera a identificação do cadastro enviada por HTTP `GET` ou HTTP `POST`. A função `getEntidade()` identifica qual classe concreta deve ser utilizada e retorna sua instância usando uma função variável. Finalmente, a função `autoload()` define como as classes do nosso sistema serão carregadas, quer dizer, como chegar ao arquivo da classe a partir do nome da classe. Apague a função `getChave()` do arquivo `functions.php` e implemente as três novas funções de acordo com a listagem a seguir:

```
<?php
function getCadastro()
{
    return (isset($_GET['cadastro']) ? $_GET['cadastro'] :
$_POST['cadastro']);
}

function getEntidade()
{
    $cadastro = getCadastro();
    $entidade = ($cadastro == 'alunos' ? 'Aluno' : 'Professor');
    $className = 'Entidade\\' . $entidade;
    return new $className();
}

function autoload($className)
{
    $fileName = str_replace('\\',DIRECTORY_SEPARATOR,$className) . '.php';
```

```
    require $fileName;
}
spl_autoload_register('autoload');
```

Vamos agora alterar o arquivo `form.php`. No lugar de importar o arquivo `entidade.php`, ele passará a importar o arquivo `functions.php`. O `entidade.php` pode ser apagado, sem medo. O papel dele foi apropriado pela família da classe `EntidadeAbstrata`. Altere o arquivo `form.php` de modo que reflita a seguinte listagem:

```
<?php
require 'functions.php';
$chave = $_GET['chave'];
$entidade = getEntidade();
$entidade = call_user_func([$entidade,
'get'],$chave);
?>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Inclusão de <?=$_GET['cadastro']?></title>
</head>
<body>
<form method="post" action="gravar.php">
Nome <input type="text" name="nome" value="<?=$entidade->getNome()?>"
autofocus="autofocus">
<input type="hidden" name="chave" value="<?=$chave?>">
<input type="hidden" name="cadastro" value="<?=$_GET['cadastro']?>">
<input type="submit" value="gravar">
</form>
</html>
```

O arquivo `form.php` envia dados para o arquivo `gravar.php`. Graças à família da classe `EntidadeAbstrata`, o arquivo `gravar.php` será consideravelmente reduzido. A gravação será feita pela invocação do método `gravar()` da classe concreta respectiva ao cadastro que está sendo manipulado. Altere o arquivo `gravar.php` de acordo com a listagem a seguir e confira a redução de linhas de código:

```
<?php
require 'functions.php';

call_user_func([getEntidade(),'gravar'],$_POST);
$cadastro = getCadastro();
header("Location:$cadastro.php");
```

A operação de remoção de um registro será igualmente reduzida no arquivo `apagar.php`, pois se limitará à chamada do método `apagar()` da classe concreta respectiva ao cadastro utilizado. Implemente essa redução alterando o conteúdo do arquivo `apagar.php` para o código-fonte a seguir:

```
<?php
require 'functions.php';
$class = getEntidade();
call_user_func([$class,'apagar'],$_GET['chave']);
$cadastro = getCadastro();
header("Location: $cadastro.php");
```

Concluímos assim a transição de uma aplicação PHP usando Programação Estruturada para uma usando Programação Orientada a Objetos. Na próxima seção, vamos ampliar o uso de Programação Orientada a Objetos, introduzindo um padrão de projeto de software, conceito que será explicado a seguir.

Só para ilustrar as mudanças feitas nesta seção, apresentamos na figura a seguir o fluxo de dados que mostra como as classes, que substituíram os arquivos `entidade.php`, `listar.php` e `pdo.php`, embora sejam usadas por mais de um arquivo, são carregadas apenas pelo arquivo `functions.php`.

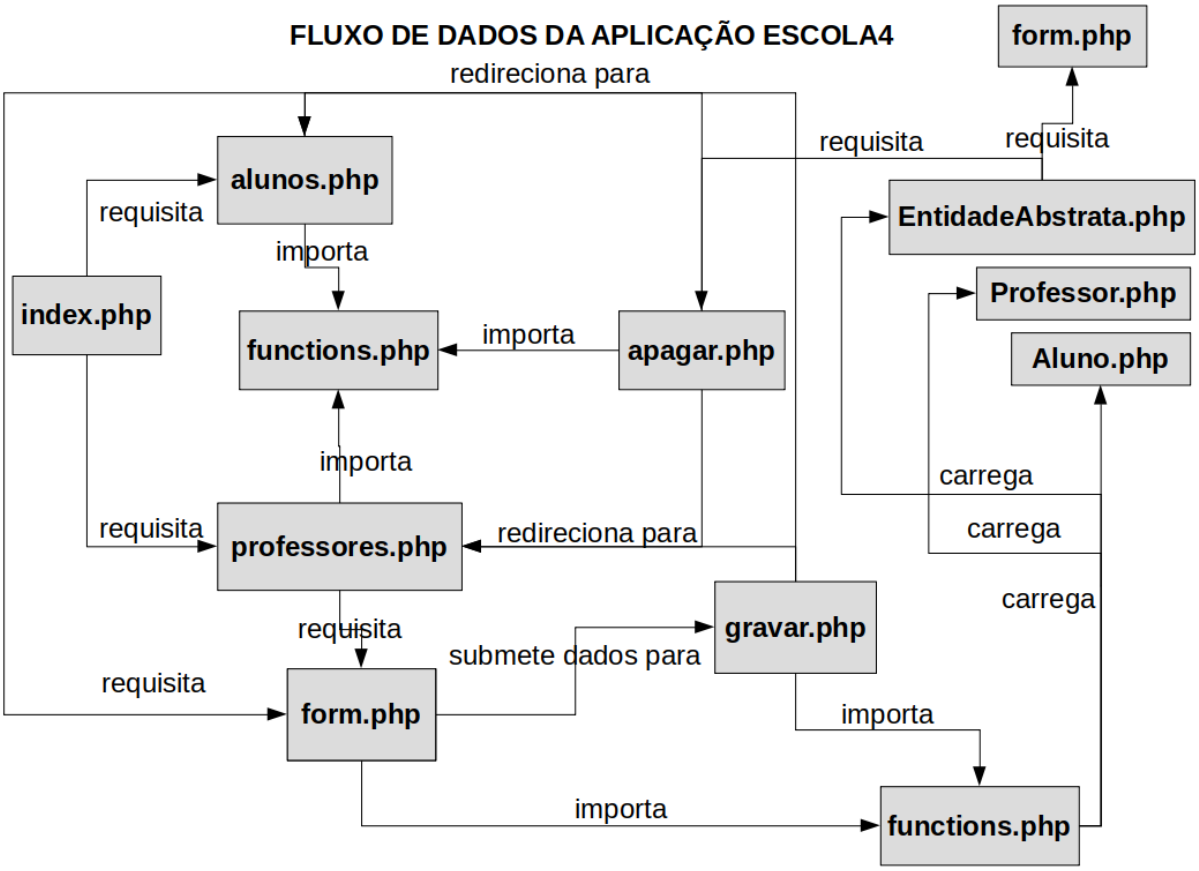


Figura 2.8: Fluxo de dados da aplicação escola4

2.5 Um cadastro com uma classe controladora de requisições

Se você já está cansado de refatorar o sistema escolar, não se preocupe, pois esta é a última seção deste capítulo. Nós encerraremos com uma introdução ao conceito de camadas e uso de padrões de projeto.

Softwares em camadas são analogias a bolos ou tortas em camadas. Para fazer um bolo grande em altura, podemos assar individualmente vários bolos e depois empilhá-los. Um software complexo, de forma análoga, pode ser construído pelo

"empilhamento" de várias "camadas", que, em um software orientado a objetos, representam conjuntos de classes que compartilham uma responsabilidade.

Um padrão de projeto, por sua vez, é uma solução para um problema recorrente em Programação Orientada a Objetos. A POO existe há mais de quatro décadas e muita experiência foi acumulada na resolução de problemas com esse paradigma de desenvolvimento de software. Essa experiência começou a ser sistematicamente documentada na forma de catálogos de soluções de problemas, ou mais precisamente de catálogos de padrões de projeto. Um padrão de projeto não implementa soluções em nenhuma linguagem de programação específica. Ele descreve de forma genérica a melhor solução encontrada para um problema recorrente de software orientado a objetos. Você pode encontrar diversos livros com catálogos de padrões de projeto que incluem implementações desses padrões em linguagens específicas, incluindo PHP.

Em nossa última refatoração, vamos dividir o sistema escolar em três camadas e em uma delas vamos implementar um padrão de projeto. A divisão em camadas que adotaremos implementará um padrão de arquitetura de software chamado Modelo-Visão-Controlador. O Modelo é a camada das regras de negócio, a implementação direta dos processos vindos do mundo real. A Visão é a interface com o usuário. O Controlador é o intermediário entre o Modelo e a Visão.

Nós já temos no projeto `escola4` uma camada de Modelo, que é o namespace `Entidade`. Agora introduziremos a camada do Controlador. Para isso, copie a pasta `escola4` como `escola5`. Dentro da pasta `escola5`, crie a pasta `Controlador`. Nessa pasta criaremos uma classe controladora, que interceptará requisições para nosso sistema. Como esta refatoração é uma introdução, nossa classe controladora não interceptará todas as requisições. Ela se chamará `ControladorEntidade` e implementará um padrão de projeto chamado *Controlador de Página*, que resolve o problema de exibir uma página HTML como resposta a uma requisição HTTP. Isso não é um problema para o protocolo HTTP, pois é justamente o que ele prevê

que aconteça, mas é um problema quando o servidor web delega a requisição para um programa no lugar de retornar um arquivo de texto.

Em suma, nossa classe controladora recebe uma requisição, interpreta, usa um objeto da camada de Modelo e exibe o resultado em uma página HTML. Para isso, crie o arquivo `ControladorEntidade.php` na pasta `Controlador`. Essa classe terá três métodos. O método `despachar()` é público e será a porta de entrada para as requisições. O método `gravar()` invocará o método de gravação de um objeto da camada de Modelo, enquanto o método `apagar()` invocará o método de remoção. Implemente a classe `ControladorEntidade` e seus métodos de acordo com a listagem a seguir:

```
<?php
namespace Controlador;

class ControladorEntidade
{
    private $requisicao = NULL;
    private $dados = NULL;

    public function despachar(array $requisicao, array $dados)
    {
        $this->requisicao = $requisicao;
        $this->dados = $dados;
        $metodo = $requisicao['metodo'];
        $this->$metodo();
    $cadastro = \getCadastro();
        header("Location: ../$cadastro.php");
    }

    private function apagar()
    {
        $class = \getEntidade();
        call_user_func([$class,'apagar'],$this->requisicao['chave']);
    }
```

```
    private function gravar()
    {
        $class = \getEntidade();
        call_user_func([$class,'gravar'],$this->dados);
    }
}
```

Observe que a classe `ControladorEntidade` utiliza algumas funções (`getCadastro()` e `getEntidade()`) presentes no arquivo `functions.php` . Não há nenhum problema em usar classes e funções em um mesmo programa PHP. A linguagem permite isso. Não é obrigatório encapsular tudo em classes. Isso deve ser feito de acordo com a necessidade. Conforme a complexidade da aplicação aumenta e mostra-se necessário atribuir um responsável para um conjunto de funções, é a hora de encapsular funções em uma classe.

A classe `ControladorEntidade` será instanciada pelo arquivo `index.php` da pasta `Controlador` . Crie esse arquivo com o código-fonte a seguir:

```
<?php
require '..' . DIRECTORY_SEPARATOR . 'functions.php';

$controladorEntidade = new Controlador\ControladorEntidade();
$controladorEntidade->despachar($_GET,$_POST);
```

É uma boa prática separar arquivos que declaram classes daqueles que as usam. Arquivos que declaram classes, de acordo com a recomendação PSR-2 do grupo PHP-FIG, devem ter o mesmo nome da classe. A mesma recomendação diz que o nome da classe deve iniciar com letra maiúscula (ou melhor, cada palavra do nome deve começar com letra maiúscula). Arquivos que não declaram classes devem iniciar com letra minúscula. Esses pequenos detalhes permitem diferenciar, em um rápido passar de olhos, arquivos de classes de arquivos de *script*.

A classe `ControladorEntidade` será usada pela classe `EntidadeAbstrata` , pois esta define os hyperlinks que geram requisições para alteração

e exclusão de registros. Por isso, vamos alterar o método `listar()` dessa classe para o conteúdo a seguir:

```
    public static function listar()
    {
        $resultSet = self::getPdo()->query('SELECT * FROM '.
static::$tabela);


        $records = $resultSet->fetchAll();

        $html = '';

        $chave = static::getChave();

    $cadastro = static::$tabela;

        foreach($records as $record)
        {
            $html .= <<<BLOCO
            <tr>
            <td>
            <a href="form.php?cadastro=$cadastro&chave={$record[$chave]}">
            {$record[$chave]}</a>
            </td>
            <td>{$record['nome']}</td>
            <td>
            <a href="Controlador\index.php?
metodo=apagar&cadastro=$cadastro&chave={$record[$chave]}">
            Excluir</a>
            </td>
            </tr>
BLOCO;
        }
        return $html;
    }
```

O atributo `action` do formulário de inclusão/alteração também passará a invocar a classe controladora. Você deve alterar a tag

`form` do arquivo `form.php` para que fique assim:

```
<form method="post" action="Controlador/index.php?metodo=gravar&cadastro=
<?=$_GET['cadastro']?>">
```

Os arquivos `index.php`, `alunos.php` e `professores.php` seguem inalterados. Os arquivos `gravar.php` e `apagar.php` podem ser apagados, pois não são mais utilizados. Suas funções foram assumidas pela classe `ControladorEntidade`. Os arquivos que restaram na raiz da pasta `escola5`, com exceção do `functions.php`, constituem a camada de Visão da aplicação. Observe que em todos eles predomina como conteúdo a linguagem HTML.

Com isso, encerramos nossa introdução ou revisão de desenvolvimento de software com a linguagem de programação PHP. Sabemos manipular bancos de dados em PHP usando a classe PDO, diretamente e encapsulada por outras classes. Sabemos utilizar classes e sabemos separar uma aplicação em camadas. No próximo capítulo passaremos a trabalhar com uma aplicação mais complexa, que faz uso dessas características.

Na figura a seguir mostramos como ficou o fluxo de dados do projeto `escola5`.

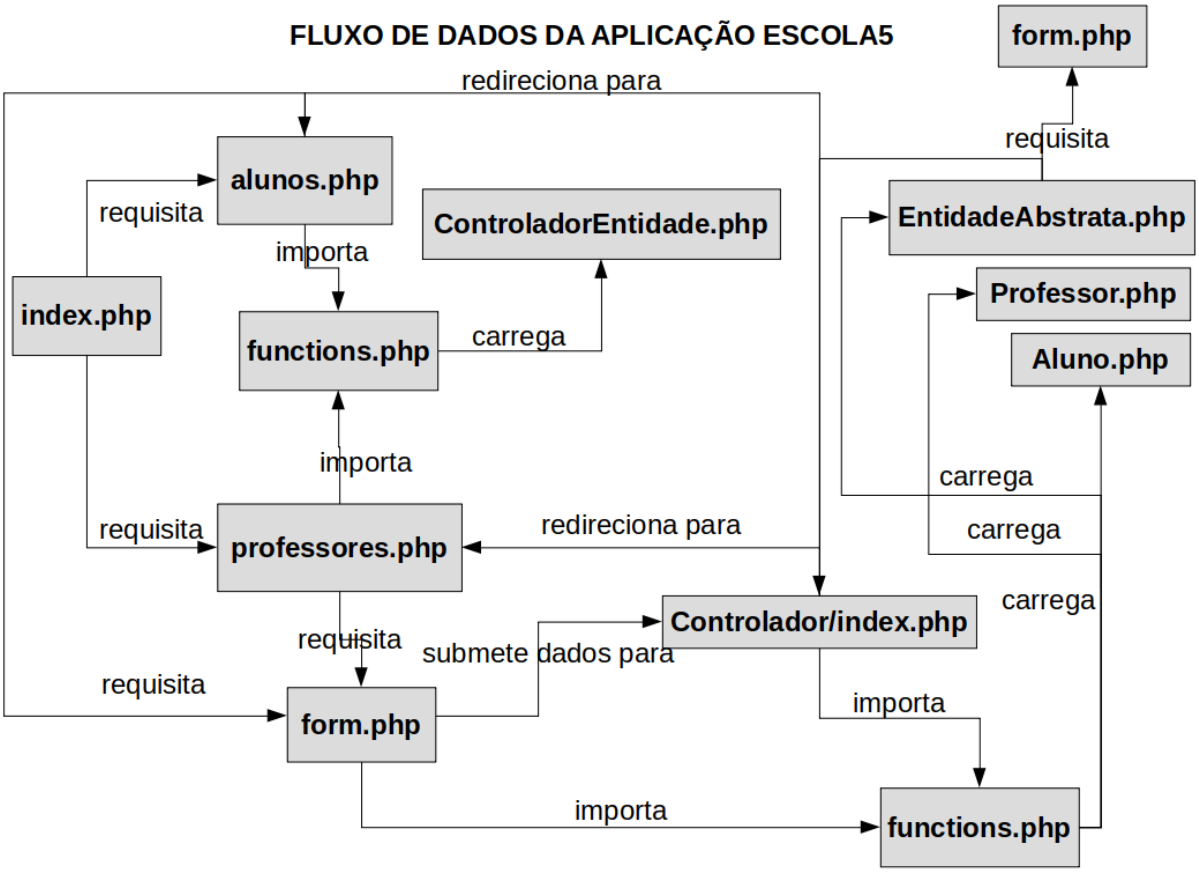


Figura 2.9: Fluxo de dados da aplicação escola5

Capítulo 3
A aplicação de exemplo

O exemplo é a escola da humanidade e só nela os homens poderão aprender. – Edmund Burke

3.1 Instalação da aplicação

Nossos exercícios serão realizados a partir de um sistema de informação com dois cadastros. O negócio dessa aplicação será esclarecido conforme a colocarmos para funcionar. A aplicação de exemplo que servirá como laboratório está disponível em https://github.com/fgsl/phpbooksample/blob/main/corps2.zip. Baixe esse arquivo e o descompacte no diretório `htdocs` do XAMPP, que é nosso diretório de trabalho.

A descompactação terá como resultado um diretório `corps2` . No terminal do seu sistema operacional, caminhe para o diretório `htdocs` do XAMPP, se você já não fez isso, e execute o interpretador PHP do XAMPP para iniciar o servidor web embutido, usando o diretório `public` da aplicação corps2 como webroot. A instrução a seguir é um exemplo de como ficaria a execução no XAMPP para Linux.

```
../bin/php -S localhost:8000 -t corps2/public/
```

A execução desse comando deverá gerar uma saída como esta no terminal:

```
[Mon Feb 10 11:08:24 2025] PHP 8.2.12 Development Server
(http://localhost:8000) started
```

O servidor web do PHP está ouvindo requisições na porta 8000. Para ter acesso à aplicação, acesse o endereço http://localhost:8000/ em

seu navegador web. O primeiro acesso deverá gerar a seguinte saída no log do servidor:

```
[Mon Feb 10 11:10:04 2025] PHP Warning:
include(/opt/lampp/htdocs/corps2/public/../vendor/autoload.php): Failed to
open stream: No such file or directory in
/opt/lampp/htdocs/corps2/public/index.php on line 23
[Mon Feb 10 11:10:04 2025] PHP Warning:  include(): Failed opening
'/opt/lampp/htdocs/corps2/public/../vendor/autoload.php' for inclusion
(include_path='.:/opt/lampp/lib/php') in
/opt/lampp/htdocs/corps2/public/index.php on line 23
[Mon Feb 10 11:10:04 2025] PHP Fatal error:  Uncaught RuntimeException:
Unable to load application.
- Type `composer install` if you are developing locally.
- Type `docker-compose run laminas composer install` if you are using
Docker.
 in /opt/lampp/htdocs/corps2/public/index.php:26
```

Se a mensagem de erro for exibida no navegador web é porque a diretiva `display_errors` do arquivo `php.ini` está configurada como On. Você pode desabilitar a exibição das mensagens no navegador alterando a diretiva para Off. Elas continuarão a ser exibidas no log do servidor no terminal.

Vamos desabilitar a exibição das mensagens de erro no navegador via aplicação. Neste momento, estamos adiantando uma boa prática de desenvolvimento de código seguro, que é não presumir que o ambiente tem as configurações adequadas. A aplicação deve desconfiar da configuração do ambiente onde está instalada e deve se assegurar de que não há brechas de segurança. A exibição de mensagens de erro para o usuário pode fornecer informações para um potencial invasor explorar vulnerabilidades.

Abra o arquivo `index.php` que está dentro do diretório `public` de corps2. Após as seguintes linhas:

```
<?php
```

```
use Laminas\Mvc\Application;
use Laminas\Stdlib\ArrayUtils;
```

Adicione a seguinte instrução:

```
ini_set('display_errors','Off');
```

A função `ini_set` do PHP modifica em tempo de execução o valor de uma diretiva do arquivo `php.ini`. Após essa alteração, se você recarregar a página no navegador, as mensagens desaparecerão, permanecendo apenas no log do servidor no terminal.

A mensagem de erro em questão, nos parágrafos de advertência, informa a ausência do diretório `vendor` e do arquivo `autoload.php`. O parágrafo final, que é o erro fatal gerado pela ausência do arquivo, orienta a execução do comando `install` do Composer.

A aplicação `corps2` utiliza componentes de terceiros, gerenciados pelo Composer. Ela não funciona neste momento porque esses componentes precisam ser instalados.

O comando `install` do Composer localiza os componentes em seus repositórios, a partir do arquivo `composer.json`, e os instala, gerando o diretório `vendor` e o arquivo `autoload.php`, que provê o carregamento das classes desses componentes. Para usar o Composer, precisamos instalá-lo. Interrompa o servidor web com a combinação de teclas CTRL+C e baixe o Composer no diretório `htdocs` a partir de https://getcomposer.org/composer.phar.

Após baixar o arquivo, entre no diretório `corps2` e execute o comando `install` do Composer, de acordo com o exemplo a seguir, em Linux:

```
../../bin/php ../composer.phar install
```

O resultado desse comando será a instalação das dependências da aplicação. Reinicie o servidor web do PHP. Observe que o comando para iniciar o servidor foi realizado a partir do diretório `htdocs`, se você for executar a partir do diretório `corps2`, tem de modificar o

caminho para o executável do PHP e o valor do parâmetro `-t`. O parâmetro `-t` indica qual será o diretório web raiz da aplicação PHP. No nosso caso, o diretório é o `public`. Se o comando php é executado dentro do diretório `public`, não é necessário informar o parâmetro `-t` porque o valor padrão é o diretório onde o comando é executado.

Ao reiniciar o servidor e recarregar o endereço http://localhost:8000/, deverá ser exibida a seguinte página:

Figura 3.1: Homepage de corps2

Isso pode deixar você mais animado, pois parece que agora a aplicação está funcionando. Só que não. Se você clicar, por exemplo, no link Setores Espaciais, terá como resultado a seguinte página:

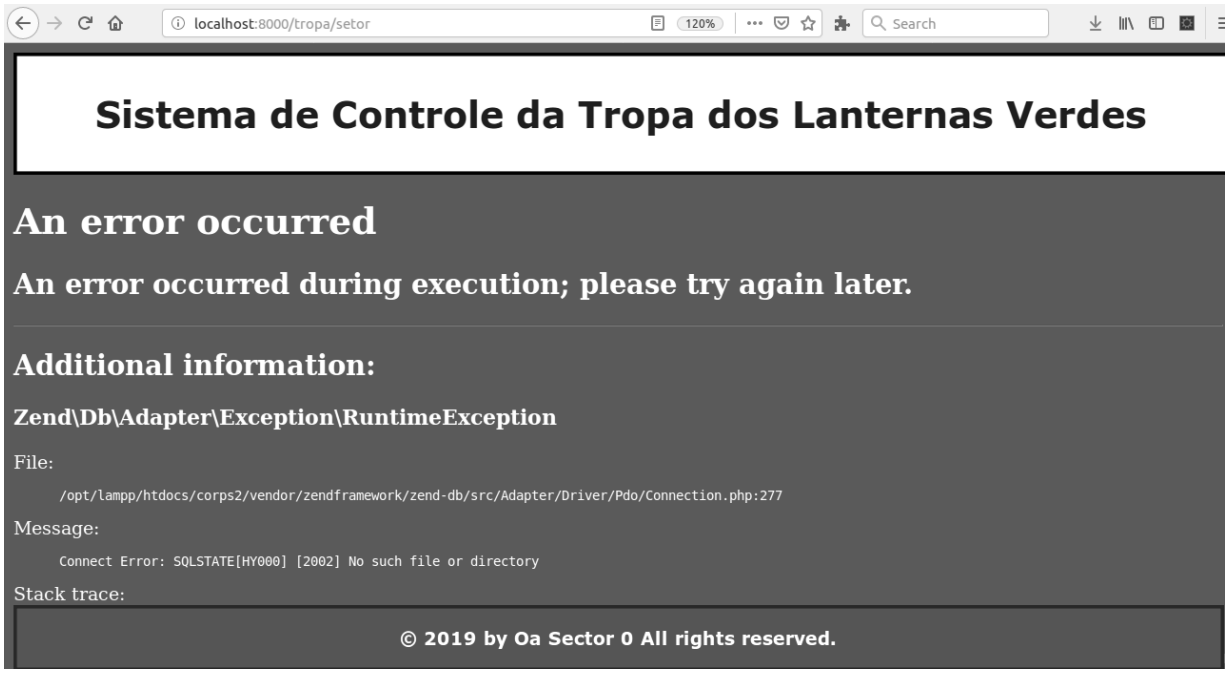


Figura 3.2: Página de erro de corps2

A exibição do erro não é uma falha da configuração. Não é o PHP que está enviando a mensagem de erro para o navegador. Essa página é produzida pela aplicação. Ela indica um erro na conexão com o banco de dados. O motivo é simples: precisamos criar o banco de dados e informar os dados de conexão corretos para a aplicação.

Vamos criar o banco de dados. Você pode usar o phpmyadmin que vem com o XAMPP para criar o banco e as tabelas. Para isso, é necessário iniciar não somente o servidor MySQL, conforme foi mencionado no primeiro capítulo, mas também o Apache. Com os dois serviços iniciados, você pode acessar o phpmyadmin por meio do endereço http://localhost/phpmyadmin/. Por padrão, o phpmyadmin no XAMPP vem instalado com um usuário administrador root sem senha.

Após acessar o phpmyadmin, clique na aba `Bancos de dados` no quadro da direita e será aberto um formulário para criação de um banco. Preencha esse formulário com corps como nome de banco e

`utf8_unicode_ci` como codificação de caracteres, conforme a figura a seguir.

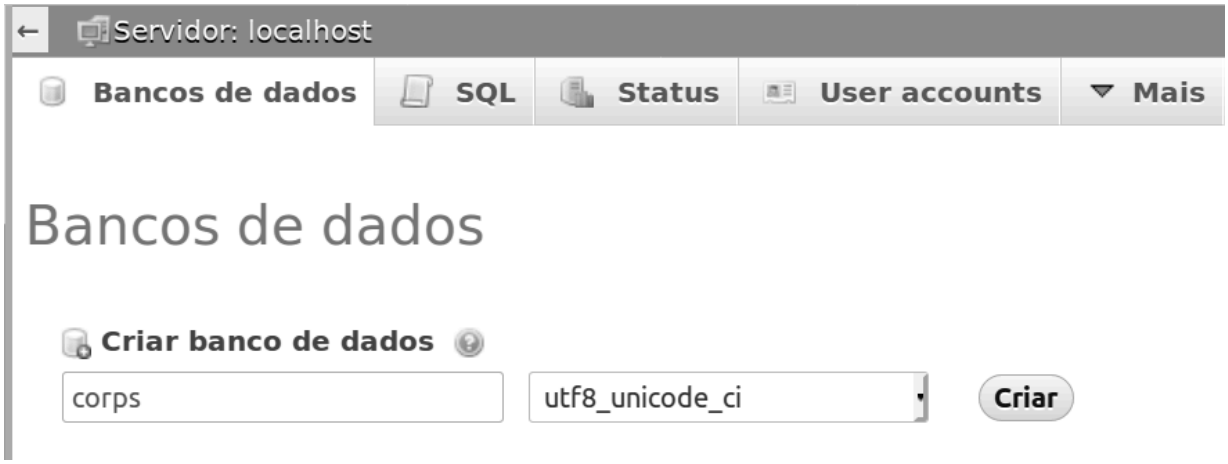


Figura 3.3: Criação de banco de dados no phpmyadmin

Após clicar no botão `Criar`, a página será atualizada e o banco corps aparecerá, tanto na estrutura em árvore no quadro da esquerda como na lista do quadro da direita. Clique no nome do banco de dados para selecioná-lo e em seguida na aba SQL no quadro da direita. O script com os comandos SQL de criação do banco de dados está disponível no GitHub. Copie para o quadro `Rodar consulta` o conteúdo do arquivo disponível em https://github.com/fgsl/phpbooksample/blob/main/sql/corps.sql, conforme a imagem a seguir:

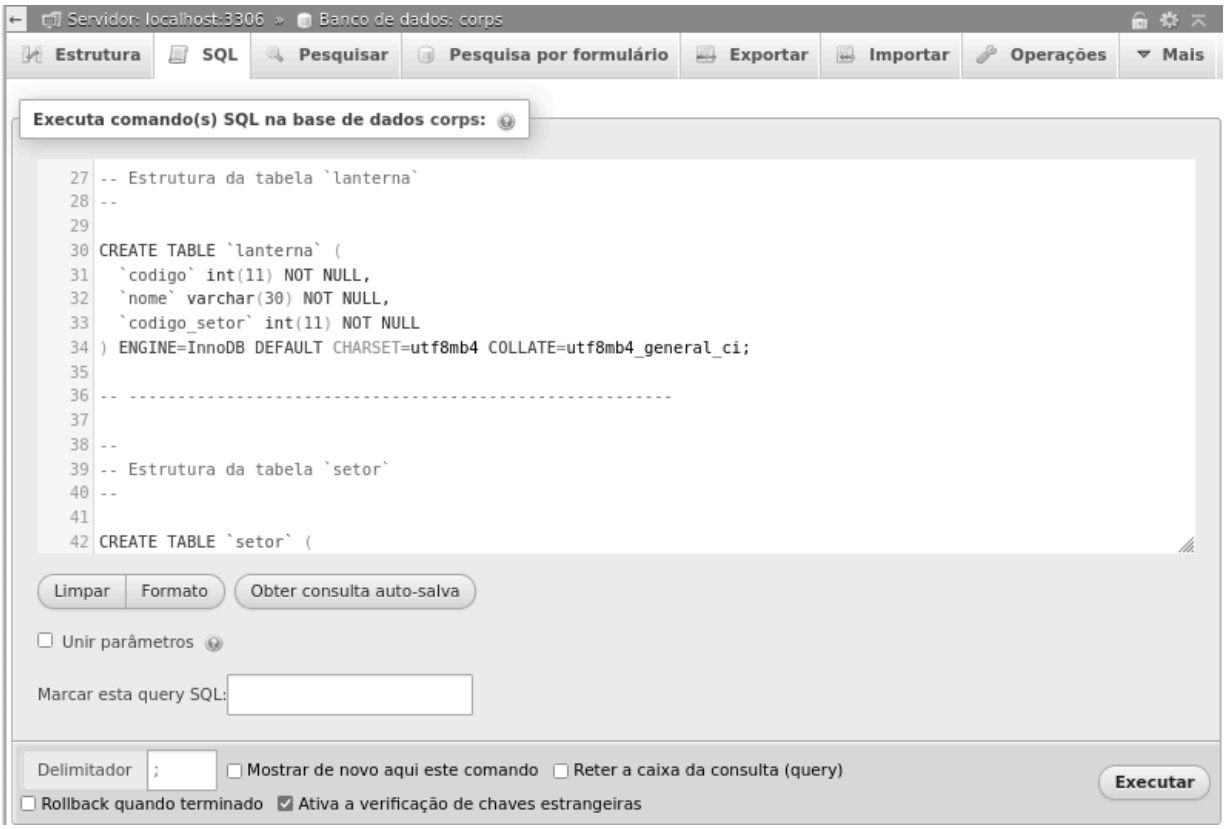


Figura 3.4: Criação das tabelas do banco corps

Após clicar no botão `Executar`, as tabelas `lanterna` e `setor`. O próximo passo é configurar a aplicação corps2 para se conectar com o banco de dados corps. Abra o arquivo `local.php` que se encontra em corps2/config/autoload. Considerando que você está usando o usuário root sem senha, o trecho que retorna o usuário e a senha de conexão com o banco deve ser alterada conforme a listagem a seguir:

```
return [
    'db' => [
        'username' => 'root',
        'password' => '',
    ],
];
```

Se você mudar a senha do usuário `root`, deverá alterar o valor da chave `password` do array. Se utilizar outro usuário, deverá alterar o

valor da chave `username`.

Recarregue a página da aplicação corps2 no navegador. Uma vez que a conexão com o banco de dados foi devidamente estabelecida, será exibido o conteúdo da figura a seguir.

Figura 3.5: Página do cadastro de setores da aplicação

Uma vez que nossa aplicação de exemplo está instalada, vamos compreender o seu negócio.

3.2 De que se trata a aplicação

O projeto corps2 consiste em um sistema de controle para uma organização fictícia chamada Tropa dos Lanternas Verdes. A Tropa é uma força policial intergaláctica compostas por operativos denominados de lanternas verdes. Os lanternas verdes são distribuídos por regiões denominadas setores espaciais.

A aplicação tem por objetivo controlar as informações sobre os lanternas em atividade e quais setores eles patrulham. A aplicação tem dois cadastros, um para os lanternas e outro para os setores.

Há um relacionamento de um para muitos entre a entidade setor espacial e a entidade lanterna verde.

As funcionalidades do projeto corps2 consistem na listagem, inclusão, alteração e exclusão de lanternas verdes e de setores espaciais. Poderia ser um sistema de controle da polícia militar, polícia civil ou polícia federal. As necessidades são similares. É preciso guardar os dados dos agentes e dos locais pelos quais eles são responsáveis.

3.3 O que falta na aplicação

Os cadastros da aplicação corps2 estão abertos: qualquer um pode incluir, alterar, excluir e listar os dados controlados pelo sistema. É necessário restringir as operações para usuários autorizados. Isso implica em implementar a autenticação dos usuários.

Nem todos os usuários de um sistema podem efetuar as mesmas operações. Após a autenticação, toda operação solicitada por um usuário deve ser submetida a um controle de acesso. Alguns usuários podem listar dados, mas não podem alterá-los nem excluí-los. É necessário criar o armazenamento para essas informações e implementar a verificação para todas as operações executadas pelos usuários.

Os dados que entram e saem da aplicação devem ser tratados e verificados. Conteúdo malicioso deve ser descartado e dados que não atendam as regras de negócio da aplicação devem ser rejeitados. É preciso implementar a conversão e filtro de dados e a validação das regras de negócio referentes aos dados.

Nossa aplicação pode ser usada diretamente por seres humanos ou indiretamente, por meio de uma outra aplicação. Para este segundo caso, é necessário que ela ofereça serviços por meio de uma API.

O suporte e a auditoria de um sistema de informação depende de saber o que está acontecendo: quem fez o quê e quando. A continuidade de serviços de uma aplicação, por usa vez, depende da flexibilidade da aplicação: a capacidade de adaptação e mudança de comportamento por meio de configuração. Isso exige a disponibilidade de alguns serviços internos.

Finalmente, o negócio de nossa aplicação corps2 é um problema internacional (ou intergaláctico), o que motiva a disponibilidade dela em vários idiomas. Para isso, é necessário que a aplicação tenha suporte à internacionalização.

Todas essas funcionalidades serão implementadas neste livro, passo a passo. Mas antes disso, nos próximos três capítulos, vamos nos aprofundar na arquitetura da aplicação, para compreender três conceitos presentes em sistemas complexos implementados com Programação Orientada a Objetos. Aqui instalamos a aplicação e entendemos o problema que ela se propõe a resolver. A partir de agora entenderemos como ela faz isso.

CAPÍTULO 4
Desenvolvimento orientado a componentes

A mais radical solução para construir software é não o construir inteiramente – Steve McConnell

Agora que a aplicação corps2 está instalada e funcional, vamos utilizá-la para ilustrar de forma prática como se faz um desenvolvimento orientado a componentes. Para nos auxiliar em nossa análise, utilizaremos o VSCode, o qual solicitamos que você instalasse no primeiro capítulo do livro.

Com o VSCode aberto, acesse o item `File -> Open Folder...` (ou `Abrir -> Nova Pasta...`, e selecione a pasta `corps2`. O VSCode abrirá a pasta e exibirá seu conteúdo conforme mostra a figura a seguir.

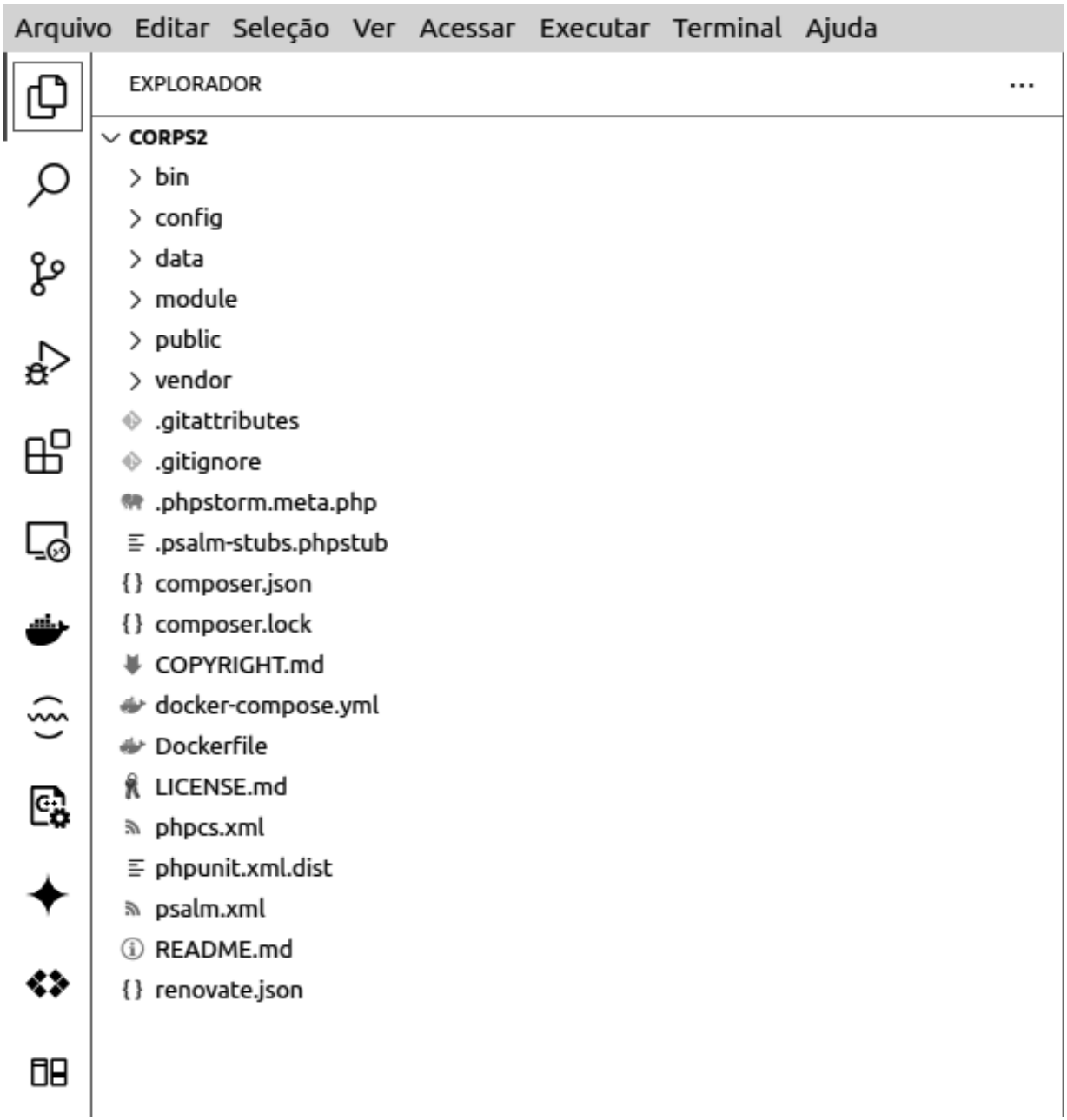


Figura 4.1: Abertura do projeto corps2 no VSCode

4.1 Usar é melhor que criar, mas nem sempre

Nossa breve discussão conceitual sobre o desenvolvimento orientado a componentes abordará apenas a sua motivação - o que nos leva a

desenvolver software a partir da integração de componentes.

A divisão de um software em componentes se faz necessária pelo controle de complexidade do software. Um programador pode dominar completamente o funcionamento de um único programa, mas não poderá conter em sua mente todos os detalhes de um grande sistema de informação com centenas de regras de negócio. É preciso dividir esse sistema de software mais complexo em pequenas partes que sejam compreensíveis independentemente, e assim possam ser mantidas por uma equipe, cujos integrantes conhecem em profundidade apenas uma parte do sistema.

Problemas complexos precisam ser divididos em pequenas partes para serem resolvidos. Essas pequenas partes em software são chamadas de componentes porque compõem o todo de um sistema completo.

Uma das difíceis decisões que um arquiteto de software deve tomar no início de um projeto é se os componentes de um sistema de controle, automação ou informação serão construídos ou comprados - ou, ainda, baixados gratuitamente. Construir um componente de software é bom quando você está aprendendo a aplicar técnicas de programação. Quando você precisa entregar um produto para um cliente, entretanto, não há muito tempo para tentativas e erros. Neste caso, usar um componente existente pode ser mais adequado. Um componente de software estável implica que um programador, programadora, ou uma equipe de programadores já passou por tentativas e erros que você teria se fosse criar o componente do zero.

Entretanto, nem sempre comprar, ou baixar, um componente pronto é a melhor solução. Há casos em que o componente pronto não tem todas as funcionalidades que o cliente deseja. Neste caso, você precisará modificar o componente existente. O trabalho de modificar algo que existe pode ser maior do que criar algo do zero, quando você precisa dominar o conhecimento embutido no componente, além das regras de negócio que obteve do cliente. Você pode pagar

o criador do componente para modificá-lo, mas isso também pode sair mais caro do que você mesmo criar. Então a resposta para saber se você criar ou compra está em fazer estimativas para os dois cenários. Isso faz parte da atividade do arquiteto de software.

Algumas funcionalidades de software são genéricas: elas se repetem para praticamente todos os projetos. Componentes que implementam esse tipo de funcionalidade são os melhores candidatos para serem reusados, descartando uma nova criação. Componentes que resolvam problemas muito específicos, por sua vez, podem trazer embutidos em si particularidades dos ambientes para os quais foram projetados, e assim não servir para ambientes com outras configurações.

Uma vez conscientes do cenário que motiva o desenvolvimento orientado a componentes e das decisões de criá-los ou comprá-los, vamos estudar como o projeto corps2 utiliza esse paradigma de desenvolvimento.

4.2 Gerenciando componentes

A princípio, desenvolver reaproveitando componentes pode parecer fácil. Você compra ou baixa gratuitamente um componente, instala-o em um determinado diretório, informa sua aplicação onde ele está e ela instancia os objetos que ele fornece. Essa é uma visão simplista do reúso de componentes.

A realidade é que um sistema de software usa tantos componentes quanto a sua complexidade exigir. A combinação entre os componentes e o núcleo do software deve produzir um sistema estável. A estabilidade não é uma garantia para qualquer combinação de componentes, o que inclui as últimas versões disponíveis deles. Isso significa que, para manter seu software estável, você poderá eventualmente que manter um componente em

uma versão mais antiga, simplesmente pelo fato de que seu sistema funciona com ela.

O controle da estabilidade do sistema de software envolve, portanto, o controle sobre as versões dos componentes utilizados. É preciso determinar qual combinação de versões mantém o sistema estável, documentar essa combinação e mantê-la.

Conforme o sistema operacional, o banco de dados e outros softwares com os quais o nosso sistema se comunica evoluem, pode ser necessário atualizar alguns componentes. Essa atualização pode gerar uma instabilidade, que pode exigir a substituição de um componente por outro.

Para o primeiro problema, no ambiente de programação PHP, temos uma solução: o Composer. O Composer é um gerenciador de dependências e é utilizado pela aplicação corps2. Você deve se lembrar que no capítulo anterior nós o utilizamos para baixar os componentes dos quais nossa aplicação depende para funcionar.

Falamos anteriormente que uma opção para criar sistemas baseados em componentes era comprar ou baixar gratuitamente componentes existentes. O Packagist é um repositório de componentes gratuitos para aplicações PHP orientadas a objeto. Em janeiro de 2019, o Packagist contava com mais de 200 mil componentes. Então, antes de pensar em criar um componente para PHP, é uma boa ideia fazer uma pesquisa em https://packagist.org/.

O Composer instala, por padrão, componentes do Packagist. A instalação é baseada em uma lista contida no arquivo `composer.json`, conforme foi mencionado no capítulo 2. Essa lista é a chave `require` do objeto JSON descrito no arquivo. Existem dois tipos de informação nessa lista. O primeiro é a versão PHP exigida para execução da aplicação. Essa informação permite impedir a instalação de uma aplicação em um ambiente com uma versão de PHP inferior à necessária. O segundo é uma versão de um componente PHP a ser instalado.

Se você abrir o `composer.json` no VSCode e procurar a chave `require`, terá uma visão das dependências diretas da aplicação, conforme mostra a figura a seguir.

Figura 4.2: Lista das dependências diretas da aplicação corps2

As dependências diretas são aquelas que nós, como programadores, sabemos que a aplicação precisa e que solicitamos ao Composer que instale. Entretanto, assim como nós construímos uma aplicação reusando componentes, os construtores de componentes podem fazê-los reusando outros componentes e assim por diante. Podem

existir, assim, dependências indiretas da aplicação, aquelas das quais os componentes dependem para funcionar. Se você abrir o arquivo `composer.lock`, verá uma relação de todas as dependências do projeto `corps2`, conforme mostra a próxima figura.

Figura 4.3: Dependências indiretas da aplicação corps2

Para o segundo problema (que a esta altura você já deve ter esquecido) que é a substituição de componentes para manutenção da estabilidade da aplicação, temos um outro auxiliar, que é o **PHP-FIG**, o grupo de interoperabilidade de frameworks PHP. Esse grupo

redige recomendações de padrões a serem seguidos no desenvolvimento de aplicações PHP orientadas a objeto, as PSRs. Se você optar por escolher componentes que implementem as PSRs, há uma boa chance de você conseguir substituí-los por outros sem a necessidade de reescrever o código que integra os componentes com a aplicação. As PSRs incluem a definição de interfaces que padronizam a comunicação entre objetos de aplicações PHP.

Os componentes utilizados pela aplicação corps2 fazem parte do projeto Laminas. Este é um dos projetos que possui representante no PHP-FIG. Como consequência, os componentes do Laminas implementam PSRs.

A figura a seguir mostra uma análise dos componentes da aplicação corps2 com relação a duas métricas: instabilidade e abstração. A instabilidade refere-se a quanto um pacote de software (que no caso do PHP refere-se a um namespace) depende de outros para funcionar. Quanto maior a dependência, maior a instabilidade, porque há mais pontos de falha. A abstração é uma medida de reúso. Quanto mais genérico for um pacote, o que implica ter mais classes abstratas e interfaces, mais reusável ele será. O mundo ideal é que os componentes fiquem o mais próximo possível da diagonal descendente, o que indica o melhor compromisso entre instabilidade e abstração.

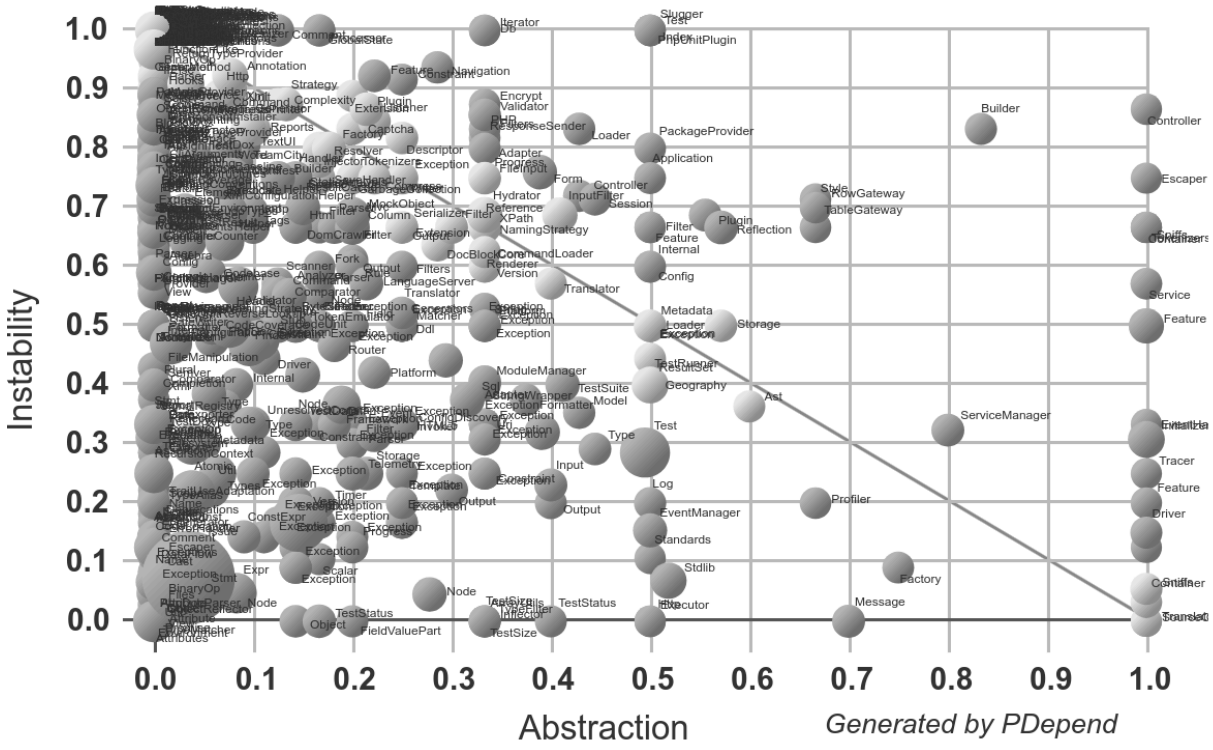


Figura 4.4: Gráfico de pacotes de corps2 com a relação instabilidade X dependência

A próxima figura apresenta uma pirâmide com várias métricas de complexidade da aplicação corps2. Ela informa que corps2 possui 418960 linhas de código (LOC) com 4933 classes (NOC) que totalizam 22440 métodos (NOM). Tanto a pirâmide quanto o gráfico anterior foram produzidos pela ferramenta *PHP Depend*, um gerador de métricas de software.

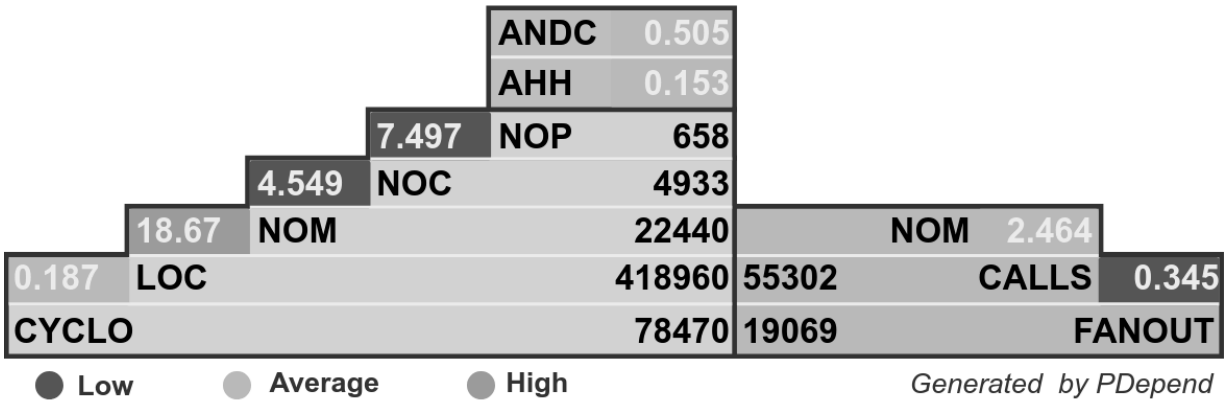


Figura 4.5: Pirâmide de complexidade de corps2

Para que você tenha uma ideia do quanto os componentes de terceiros participam da aplicação, saiba que corps2 tem apenas 9 classes que foram criadas especificamente para ela. As demais 4924 classes são dos componentes utilizados pela aplicação. Em corps2 temos 99% de reúso de código existente.

Após termos um exemplo prático de desenvolvimento baseado em componentes, no próximo capítulo nos aprofundaremos mais na arquitetura da aplicação corps2 abordando o paradigma do desenvolvimento orientado a eventos.

Capítulo 5
Desenvolvimento orientado a eventos

Uma forma um pouco menos comum, mas ainda muito útil para uma função de argumento único é um evento – Robert C. Martin

Quando clicamos no hyperlink **Setores Espaciais** na página inicial, somos movidos para o endereço tropa/setor, o qual apresenta uma lista de setores espaciais — se eles tiverem sido cadastrados. O evento de clique, interceptado pelo navegador, produz uma requisição HTTP do tipo `GET`. A requisição, recebida pela aplicação, produz uma página HTML. Neste capítulo estudaremos qual foi o caminho trilhado pela aplicação para produzir essa página e, ao longo dele, compreenderemos como ela faz uso de desenvolvimento orientado a eventos.

O primeiro detalhe a ser notado é que o endereço tropa/setor não corresponde a um caminho de diretório do sistema de arquivos. Ele é um endereço virtual, interpretado pela aplicação para gerar uma resposta HTTP. Vamos dar um *spoiler* aqui, a página exibida como resposta à requisição ao endereço tropa/setor é uma combinação de três arquivos: `corps2/module/Application/view/layout/layout.phtml`, `corps2/module/Tropa/view/tropa/setor/index.phtml` e `corps2/module/Tropa/view/setores.phtml`. Você deve concordar que qualquer um desses caminhos é mais complicado de lembrar do que tropa/setor. É por isso que um endereço como esse é classificado como URL amigável.

A combinação de três arquivos para formar uma única página HTML está encapsulada em uma única instrução no arquivo `index.php` da pasta `public`:

```
$app->run();
```

A variável `$app` é uma instância da classe `Application`. Esse objeto já está populado com os dados de configuração da aplicação. A partir

desse objeto, é chamado o método `run`. Ele não traz em suas instruções nada explícito sobre como o endereço tropa/setor gerou uma combinação de conteúdo de três arquivos. Na verdade, o método `run` não participa diretamente da criação da página de resposta. Ele apenas dispara eventos.

A aplicação corps2 define três eventos possíveis: `ROUTE`, `DISPATCH` e `FINISH`. O caminho feliz da aplicação prevê que os três sejam disparados nessa ordem. Primeiro, a aplicação encontra uma rota (`ROUTE`) do URL para uma classe controladora que vai processar a requisição. Depois, um objeto da classe controladora executará o método solicitado na requisição - esse é o despacho (`DISPATCH`) da requisição. Finalmente, a aplicação encerra (`FINISH`) o processamento da requisição, gerando um conteúdo que será enviado em resposta.

Vamos nos debruçar sobre o evento `ROUTE` como um exemplo de programação orientada a eventos. O método `run` configura um objeto, identificando o evento `ROUTE`:

```
$event->setName(MvcEvent::EVENT_ROUTE);
```

Um objeto da classe `EventManager` (`$events`) dispara o evento configurado com o método `triggerEventUntil`:

```
$result = $events->triggerEventUntil($shortCircuit, $event);
```

Disparar um evento significa que um objeto está avisando outros sobre uma mudança de estado e pedindo que eles façam alguma coisa. Os objetos avisados são denominados de ouvintes (*listeners*) porque ficam esperando um "ruído" para agir. O "ruído" é o evento.

O objeto gerenciador de eventos (`$events`, neste caso), mantém uma coleção de ouvintes. Os ouvintes dessa coleção têm de ser definidos antes dos eventos serem disparados. O importante é saber que as ações a serem executadas quando um evento é disparado podem ser alteradas pela adição ou remoção de ouvintes da coleção do gerenciador de eventos.

Os ouvintes da aplicação `corps2` são carregados pelo método `init` da classe `Application`, no arquivo `config/container.php`:

```
return Application::init($appConfig)
    ->getServiceManager();
```

Os ouvintes são definidos nos arquivos de configuração, que estão mesclados no array da variável `$appConfig`. No trecho do método `Application::init` a seguir, a chave `listeners` é procurada na primeira dimensão do array de configuração e na configuração do objeto em `$serviceManager`.

```
$listenersFromAppConfig     = $configuration['listeners'] ?? [];
$config                     = $serviceManager->get('config');
$listenersFromConfigService = $config['listeners'] ?? [];
```

Uma funcionalidade pode ser habilitada ou desabilitada rapidamente dessa forma, o que é extremamente conveniente para a manutenção da aplicação.

Mas o que isso tem a ver com a página apresentada como resposta à requisição tropa/setor? Tudo. O roteamento desse URL para a página de listagem dos setores é realizado pelo disparo de um evento. O componente `Laminas\Mvc` provê um `listener` chamado `RouteListener`, que executa o método `onRoute` quando o evento `ROUTE` é disparado. O método `onRoute` procura uma classe controladora que assuma a requisição ao URL tropa/setor e, a partir do que está configurado na chave `router` do arquivo `module.config.php` do módulo `Tropa`, ele descobre que deve delegar o processamento para o método `indexAction` da classe `SetorController`.

Na verdade, o método `indexAction` é chamado indiretamente, pelo método `onDispatch`. Aqui é aplicada uma convenção de código: o prefixo "on" em um método, no Laminas, implica um método que responde a um evento - que é a palavra seguinte ao prefixo. Assim, o método `onDispatch` é o método que trata o evento `DISPATCH`. A escolha de nomes adequados facilita o reconhecimento do comportamento das classes em um software orientado a objetos.

O método `onDispatch` não é chamado por acaso. Em um momento anterior, de configuração, ele foi registrado como método ouvinte pela classe `EventManager` pelo método `attach`. O controlador herda essa configuração pelo método `attachDefaultListeners` da classe `AbstractController` do framework:

```
    protected function attachDefaultListeners()
    {
        $events = $this->getEventManager();
        $events->attach(MvcEvent::EVENT_DISPATCH, [$this, 'onDispatch']);
    }
```

De forma resumida, a aplicação corps2 utiliza o componente `EventManager` para implementar uma lógica orientada a eventos no processamento das requisições, a qual dispensa o uso de estruturas de decisão e a execução desnecessária de instruções.

Neste capítulo usamos uma implementação existente para ilustrar o desenvolvimento orientado a eventos, mas haverá oportunidade de implementar uma nova funcionalidade orientada a evento quando abordarmos a autenticação de usuários.

CAPÍTULO 6
Injeção de dependências

A injeção de dependência é um padrão extremamente útil em qualquer aplicação não trivial - Jeremy Clark

No capítulo 3 nós abordamos o desenvolvimento orientado a componentes, mas não definimos o que seria um componente. Isso não deve ter prejudicado a leitura e a compreensão do capítulo, pela intuição de que componente é uma parte de algo. Mas agora precisamos refletir especificamente sobre componente para o contexto do software. Fowler (2004) chama de componente "uma série de software que se destina a ser usado, sem alteração, por um aplicativo que está fora do controle dos escritores do componente".

A definição de Fowler traz uma sutileza: o uso sem alteração. A injeção de dependência trata de garantir essa característica. A palavra "dependência" implica que um software depende de um componente. De forma mais atômica, uma classe do software depende de uma classe do componente. O comportamento do software pode ser modificado se a dependência - o componente do qual ele depende - for substituído. Essa substituição pode ser feita de um modo complicado - alterando o código-fonte da classe que usa a dependência - ou de um modo simples - sem alterar o código-fonte da classe. Essa segunda opção está mais de acordo com a definição de Fowler e é a ideal.

Para que seja possível modificar uma dependência sem alterar a classe da qual ela depende, a dependência não pode estar fortemente acoplada à classe, como se estivesse gravada em pedra, mas ser *injetada*, como se fosse uma tomada que se pode conectar e desconectar com facilidade.

Como uma ilustração introdutória, vamos imaginar uma classe chamada `ContaCorrente`. A classe `ContaCorrente`, em seu projeto, tem

a responsabilidade de enviar mensagens SMS. Poderíamos implementar essa funcionalidade como um método da classe, mas já existem componentes que fazem isso. Suponhamos que um deles fornece uma classe chamada `EmissorDeSMS`. Um modo de estabelecer uma dependência entre essas duas classes é implementar um acoplamento forte. Isso implica instanciar a classe `EmissorDeSMS` dentro da classe `ContaCorrente`, conforme a listagem a seguir:

```
use Componente\EmissorDeSMS;

class ContaCorrente
{
    private ?EmissorDeSMS $emissorDeSMS = null;

    public function __construct()
    {
        $this->emissorDeSMS = new EmissorDeSMS();
    }
}
```

Ao instanciar internamente a classe `EmissorDeSMS`, a classe `ContaCorrente` pode dispor dos métodos da primeira para enviar mensagens SMS. Mas e se, por alguma razão, precisarmos substituir a classe `EmissorDeSMS` por outra? As razões pela troca podem variar desde a busca por melhor desempenho (um algoritmo otimizado) até pelo valor agregado (funcionalidades que não existem na classe atual). Independente da motivação pela mudança, a questão é que, da forma como a classe `EmissorDeSMS` foi associada à classe `ContaCorrente`, o único modo de mudar o acoplamento entre as duas é alterar o código-fonte da classe `ContaCorrente`. Por exemplo, se a nova e melhor classe se chamar `EmissorDeSMSMaisPoderoso`, teríamos de alterar pelo menos duas linhas de código, conforme a listagem a seguir:

```
use ComponenteMelhor\EmissorDeSMSMaisPoderoso;

class ContaCorrente
{
```

```
    private ?EmissorDeSMSMaisPoderoso $emissorDeSMS = null;

    public function __construct()
    {
        $this->emissorDeSMS = new EmissorDeSMSMaisPoderoso();
    }
}
```

A forma como a implementação da dependência entre as classes foi feita gerou um forte acoplamento entre elas. Se for necessária uma nova substituição da dependência, haverá novamente mudança no código-fonte da classe `ContaCorrente`. Como seria maravilhoso se pudéssemos trocar a dependência sem alterar a classe `ContaCorrente` ... Mas nós podemos! Basta usar a técnica da injeção de dependência na implementação da classe.

Vamos voltar no tempo e fazer a primeira versão da classe `ContaCorrente` com uma implementação diferente, que não mencione diretamente a dependência `EmissorDeSMS`:

```
use Padroes\InterfaceEmissorDeSMS;

class ContaCorrente
{
    private ?InterfaceEmissorDeSMS $emissorDeSMS = null;

    public function __construct(InterfaceEmissorDeSMS $emissorDeSMS)
    {
        $this->emissorDeSMS = $emissorDeSMS;
    }
}
```

Observe que agora o atributo `$this->emissorDeSMS` não recebe uma instância de classe criada pelo operador `new`, mas uma instância de classe contida em uma variável. Essa variável tem um tipo explicitamente declarado: `InterfaceEmissorDeSMS`. Como o nome sugere, não é uma classe, é uma interface. O código dessa interface, no nosso exemplo, é este:

```php
namespace Padroes;

interface InterfaceEmissorDeSMS
{
    public enviarMensagem(string $mensagem): bool;
}
```

Isso significa que o objeto esperado pelo método construtor da classe `ContaCorrente` deve implementar o método `enviarMensagem`, o qual por sua vez tem um argumento obrigatório. Com o uso de uma interface, o acoplamento entre a classe `ContaCorrente` e sua dependência é feito no momento de criação da instância da primeira, assim:

```php
$emissorDeSMS = new EmissorDeSMS();
$contaCorrente = new ContaCorrente($emissorDeSMS);
```

Agora estamos injetando a dependência `EmissorDeSMS` na classe `ContaCorrente`. Se quisermos que `ContaCorrente` use uma instância de `EmissorDeSMSMaisPoderoso`, não teremos de modificar uma linha sequer da classe. Basta injetar outra instância:

```php
$emissorDeSMS = new EmissorDeSMSMaisPoderoso();
$contaCorrente = new ContaCorrente($emissorDeSMS);
```

Estamos assumindo que `EmissorDeSMSMaisPoderoso` implementa a interface `InterfaceEmissorDeSMS` - o que implica ter uma implementação do método `enviarMensagem`. Para a classe `ContaCorrente`, não importa qual é a classe do objeto injetado na variável `$emissorDeSMS`, desde que implemente a interface especificada como tipo de argumento do construtor. A interface estabelece um padrão e torna a classe `ContaCorrente` independente de uma implementação específica.

Se for necessário trocar a dependência para outra classe, por exemplo, `EmissorDeSMSMaisPoderosoAinda`, a classe `ContaCorrente` continuará intocada. Só precisamos mudar o objeto injetado no construtor:

```
$emissorDeSMS = new EmissorDeSMSMaisPoderosoAinda();
$contaCorrente = new ContaCorrente($emissorDeSMS);
```

O exemplo da emissão de SMS para conta corrente serviu para introduzirmos a técnica da injeção de dependência. Mas ele é um exemplo simplificado. Veremos a partir de agora exemplos de injeção de dependências mais elaborados, primeiro no código-fonte do projeto corps2 e depois no código-fonte do Laminas, utilizado pelo projeto.

6.1 Injeção de dependência no controlador

A aplicação corps2 faz uso de diversos padrões de projeto, que são elementos reutilizáveis em software orientado a objetos. Um desses padrões é Controlador de Páginas, que define uma classe responsável por exibir uma página web de acordo com os dados da requisição HTTP.

A classe `SetorController` é uma implementação de um Controlador de Páginas. Mas ela depende de outras classes para funcionar: uma para mapear a tabela onde serão gravados os dados manipulados pelo controlador e outra para gerenciar a sessão do usuário. O acoplamento das dependências é feito no construtor da classe `SetorController`.

```
public function __construct(SetorTable $table, SessionManager
$sessionManager)
{
    $this->table = $table;
    $sessionManager->start();
}
```

O ideal é que a injeção de dependências seja feita por interfaces e não por implementações. No caso do construtor da classe `SetorController`, os argumentos precisam ser de classes específicas. Neste caso, não há independência de classes, apenas de configuração. Embora `SetorController` aceite objetos de classes específicas, a configuração desses objetos pode ser alterada antes de eles serem injetados no controlador. A criação desses objetos é feita no arquivo de configuração `module.config.php`, por meio de uma função anônima que atua como uma fábrica de objeto:

```
Controller\SetorController::class => function($sm){
    $table = $sm->get(SetorTable::class);
    $sessionManager = new SessionManager();
    return new SetorController($table, $sessionManager);
},
```

Os argumentos `$table` e `$sessionManager` são criados e injetados no arquivo de configuração. Nesse arquivo, podemos manipular os objetos antes de injetá-los na instância de `SetorController`, afetando seu comportamento.

6.2 Injeção de dependência no mapeador de tabelas

Você deve ter reparado que o objeto mapeador da tabela, na fábrica de instância de `SetorController`, não é criado com o operador `new`, mas com a chamada a um método `get` que recebe como argumento o nome da classe `SetorTable`. Vamos entender o que está acontecendo.

O método `get` está sendo chamado de uma instância da classe `ServiceManager`, que no Laminas é a responsável por intermediar a criação de objetos usando fábricas. O argumento passado para `get` é uma chave de localização para uma função anônima ou uma classe que implementa a criação de um objeto. `ServiceManager` separa a

complexidade da criação de instâncias de uma classe do local onde elas precisam ser usadas. Isso evita que a cada vez que uma instância de uma classe é necessária tenhamos de repetir todas as linhas de código necessárias para sua criação.

Geralmente, quando usamos uma fábrica para criar um objeto, essa criação não é feita usando somente o operador `new`. Fábricas escondem cadeias de dependências. Uma cadeia de dependência é a situação na qual um objeto, para ser criado, precisa de outro, que precisa de outro e assim por diante. O "assim por diante" pode ser a última instância de classe ou pode ser uma série de outras instâncias. A fábrica faz com que essa complexa sequência de criações e injeções de dependência seja escondida de quem vai usar o objeto final.

Vamos esclarecer isso compreendendo como a instância de `SetorTable` é criada pelo método `get` de `ServiceManager`. No trecho de código a seguir, temos duas funções anônimas que funcionam como fábricas. A primeira devolve uma instância de `SetorTable`, que para ser criada depende de um objeto referenciado pela variável `$tableGateway`. A segunda função cria o objeto atribuído a `$tableGateway`. Quando a fábrica do controlador `SetorController` usa o método `get` de `ServiceManager` para obter uma instância de `SetorTable`, ela ignora a criação dos quatro objetos nas duas fábricas do mapeador de tabelas.

```
SetorTable::class =>  function($sm) {
    $tableGateway = $sm->get('SetorTableGateway');
    $table = new SetorTable($tableGateway);
    return $table;
},
'SetorTableGateway' => function ($sm) {
    $dbAdapter = $sm->get('Laminas\Db\Adapter');
    $resultSetPrototype = new ResultSet();
    $resultSetPrototype->setArrayObjectPrototype(new Setor());
    return new TableGateway('setor', $dbAdapter, null,
$resultSetPrototype);
},
```

Assim, a injeção de dependências, combinada com as fábricas de objetos, evita não somente a alteração de algumas classes do sistema como esconde a complexidade de criação de objetos que têm cadeias de dependências.

Ao longo deste livro veremos outros exemplos de injeção de dependências usando componente do Laminas. Com este capítulo, encerramos uma sequência importante de fundamentação sobre arquitetura de software aplicada para Programação Orientada a Objetos. Daqui em diante partiremos para a implementação e modificação de funcionalidades do projeto corps2 com o uso de componentes prontos.

CAPÍTULO 7
Segurança de aplicações web

O maior inimigo do homem é a segurança. – William Shakespeare

Neste capítulo, concentramos as orientações gerais sobre desenvolvimento de código seguro. Mostramos aqui alguns exemplos pontuais de implementação de código defensivo usando Laminas e indicamos os capítulos específicos sobre determinados tópicos de segurança em aplicações de software, como injeção de SQL, filtros, validadores, autenticação e controle de permissões.

De acordo com Howard e Leblanc (2005, p. 25), um software seguro não é "um código de segurança ou um código que implementa os recursos de segurança", mas, sim, "um código projetado para suportar ataques por invasores mal-intencionados. Um código seguro também é um código robusto".

O objetivo das orientações deste capítulo, e dos capítulos indicados por ele, é o de fazer com que o código de sua aplicação resista a ataques sem esperar que o ambiente a proteja. Observe que no parágrafo anterior dissemos que o código seguro é escrito para suportar ataques, e não para evitá-los. Não conseguimos impedir que pessoas mal-intencionadas tentem explorar vulnerabilidades, mas podemos dificultar o seu trabalho.

7.1 Tratamento e neutralização de saída perigosa

Uma função crítica para scripts de visão executarem é a neutralização apropriada de saída potencialmente perigosa, como forma de rechaçar ataques, tais como *Cross-Site Scripting*.

O termo original para isso em inglês é *escaping*, que literalmente remete à "fuga" ou "evasão". Em inglês, a ideia consiste em que o interpretador "fuja" de caracteres especiais que têm significado de comandos em determinados contextos, como por exemplo os caracteres < e > , que são delimitadores HTML.

Um *output escaping* é uma operação de neutralização da interpretação do texto como um conjunto de comandos a serem executados. Por meio de transformações adequadas, o texto será exibido como foi recebido. No caso dos delimitadores, os caracteres `<` e `>` serão substituídos respectivamente por < e > .

Uma boa prática é sempre tratar variáveis quando as enviar para a saída, a menos que você esteja usando uma função ou um método ou auxiliar que execute esse passo por si próprio.

```
<?php
// Má prática de script de visão:
echo $this->variable;
// Boa prática de script de visão:
echo $this->escapeHtml($this->variable);
```

A classe `Laminas\Escaper\Escaper` usa o método `escapeHtml()` para neutralizar conteúdo HTML de modo que tags HTML e seus atributos não sejam interpretados, exibindo o texto exatamente como ele foi recepcionado. O método `escapeHtml()` pode ser chamado a partir da referência `$this` nos arquivos `phtml` graças à *view helper* `Laminas\View\Helper\EscapeHtml` .

O construtor da classe `Escaper` recebe como argumento a codificação de caracteres. Quando você utiliza o método a partir da *view helper*, ele está configurado com essa codificação por padrão.

- Para neutralizar um atributo HTML, use o método `escapeHtmlAttr()` .
- Para neutralizar um código JavaScript, use o método `escapeJs()` .

- Para neutralizar um código CSS, use o método `escapeCss()`.
- Para neutralizar um código que faz parte de um URL, use o método `escapeUrl()`.

7.2 Ataques XSS

XSS é a sigla para *Cross-Site Scripting*. O X no lugar do C se deve à utilização da sigla CSS para as folhas de estilo em cascata (*Cascading Style Sheets*), e ao fato de a letra remeter a cruzamento em inglês. Como esses ataques são uma injeção de HTML, CSS ou código de script em uma página, JavaScript é uma ameaça em especial; sua causa primária é exibir dados mal interpretados pelo navegador.

Filtros são geralmente usados para tratar elementos HTML, ajudando a evitar tais ataques. Se, por exemplo, um campo de formulário é populado com dados não confiáveis (e qualquer entrada de dados do lado de fora da aplicação é não confiável!), então esse valor não deve conter HTML, ou, se contiver, então esse HTML tem de ser tratado. Chamar o método `filter()` sobre qualquer objeto `Laminas\Filter\*` executa transformações sobre a entrada de dados. Um exemplo com os caracteres `&` e `"` é dado a seguir.

```
<?php
use Laminas\Filter\HtmlEntities;
require 'vendor/autoload.php';
$htmlEntities = new HtmlEntities();
echo $htmlEntities->filter('&'); // &amp;amp;
echo $htmlEntities->filter('"'); // &amp;quot;
```

O filtro `HtmlEntities` garante que os caracteres sejam exibidos exatamente como foram passados para o filtro. Uma vez que eles fazem parte da sintaxe do HTML, seriam interpretados pelo navegador, mas o filtro os converte no código correspondente à sua

representação gráfica. No próximo capítulo trataremos especificamente de filtros.

7.3 Ataques de injeção de SQL

Ataques de injeção de SQL (SQL injection) são aqueles nos quais as aplicações são manipuladas para retornar dados para um usuário que não tem o privilégio para acessar/ler tais dados. Esse acesso geralmente é realizado por uma tentativa aleatória de um atacante de explorar uma falha, conhecida ou não, no código envolvendo consultas SQL que usam variáveis PHP (que exige apóstrofes).

O desenvolvedor deve levar em conta que essas variáveis podem conter símbolos que resultarão em código SQL incorreto – por exemplo, o uso de uma apóstrofe no nome de uma pessoa. O ataque é realizado por uma pessoa mal-intencionada que detecta tais falhas e as manipula, por exemplo, pela especificação de um valor para uma variável PHP por meio do uso de um parâmetro HTTP ou outro mecanismo similar.

Tente descobrir onde está o erro e a consequente falha de segurança no código a seguir:

```
$name = $_GET['Name'];
//$_GET['Name'] = "Maggie O'Farrell";
$sql = "SELECT * FROM books WHERE author = '$name'";
echo $sql;
// SELECT * FROM books WHERE author = 'Maggie O'Farrell'
```

O valor procurado contém um caractere que funciona como delimitador de texto. Neste caso, ocorrerá um erro e nenhum dado será retornado. Um atacante, entretanto, pode modificar a condição de filtro para obter mais dados, assim:

```
$name = $_GET['Name'];
//$_GET['Name'] = "X' and '' = '";
```

```
$sql = "SELECT * FROM books WHERE author ='$name'";
echo $sql;
// SELECT * FROM books WHERE author = 'X' and '' = ''
```

Entendeu o que aconteceu? O atacante injetou uma expressão com uma tautologia (algo sempre verdadeiro), para que a consulta não retorne apenas o nome de um autor, mas de todos.

`Laminas\Db` tem algumas funcionalidades internas que ajudam a mitigar tais ataques, embora não sejam substitutas para a boa programação.

Quando se insere dados de um array, os valores são inseridos como parâmetros, por padrão, reduzindo o risco de alguns tipos de problemas de segurança.

Consequentemente, você não precisa aplicar tratamento ou cerceamento por apóstrofes para valores em um array de dados. A implementação de Laminas para tratamento de SQL Injection será conferida em uma seção específica mais adiante.

7.4 Ataques de simulação de requisição

Ataques *Cross-Site Request Forgery* (CSRF) empregam um caminho quase oposto aos ataques *Cross-Site Scripting* (XSS). Enquanto ataques XSS partem de um suposto website confiável, ataques CSRF partem de supostos usuários confiáveis.

Um ataque CSRF consiste na criação de um formulário falso, mas que é aceitável pela aplicação. Ele é submetido de modo a obter ou modificar dados. Por ter sido forjado, não possui as validações esperadas do lado cliente. Ataques CSRF podem afetar atividade potencialmente perigosas, como alteração de senhas (que geralmente são feitas por formulários).

Para se proteger contra ataques CSRF originários de sites de terceiros, você deve adotar algumas práticas, que descrevemos a seguir.

Antes de fazer qualquer tipo de atividade "perigosa", tal como alterar a senha ou comprar um carro, isso requer que o usuário se reautentique de modo que ele seja forçado a interagir com o website (usando `Laminas\Authentication`, por exemplo). Pela desabilitação de páginas automatizadas e pelas submissões de formulários, a janela que um atacante tem para iniciar o ataque é severamente limitada.

Use identificadores (IDs) com hash em formulários submetidos, pois isso pode reduzir o dano de um ataque CSRF. Uma vez que um único hash tenha sido usado para servir uma requisição, ele é subsequentemente invalidado, e qualquer requisição usando esse ID não é honrada.

A segunda opção pode ser implementada com o uso de `Laminas\Form\Element\Csrf`. Esse componente provê proteção contra ataques CSRF a partir de formulários, garantindo que os dados foram submetidos pela sessão do usuário que gerou o formulário e não por um script farsante. Ele adiciona um hash ao formulário e verifica-o quando o formulário é submetido.

A seguir, mostramos duas formas de adicionar um elemento CSRF a um formulário gerado pela classe `Laminas\Form\Form`.

Adicionando `Csrf` com a instância da classe:

```
use Laminas\Form\Element;
use Laminas\Form\Form;
$csrf = new Element\Csrf('csrf');
$csrf->setCsrfValidatorOptions(['timeout' => 600]);
$form = new Form('secure-form');
$form->add($csrf);
```

Usando `Csrf` com definição por array:

```
use Laminas\Form\Form;
$form = new Form('secure-form');
$form->add([
    'type' => 'Laminas\Form\Element\Csrf',
    'name' => 'csrf',
    'options' => [
        'csrf_options' => [
           'timeout' => 600
        ]
    ]
]);
```

Você poderá verificar se o formulário recebido foi forjado utilizando o método `isValid()` de `Laminas\Form\Form`.

Além disso, temos de tratar os campos do formulário. Filtros (tais como os disponíveis em `Laminas\Filter`) são geralmente usados para tratar elementos HTML. Se, por exemplo, um campo de formulário é populado com dados não confiáveis (e qualquer entrada de dado de fora da aplicação é não confiável!), então esse valor não deve conter HTML (remoção), ou, se contiver, então tem esse HTML tratado.

Estranho... Parece que você já viu esse parágrafo em algum lugar... Ei, é isso mesmo! Você viu o mesmo parágrafo na seção de Ataques XSS. Viu? Sua mente já está fazendo as ligações.

> **Nota**
>
> Pode ocorrer, algumas vezes, de a saída HTML precisar ser exibida – por exemplo, quando usamos um editor HTML baseado em JavaScript; nesses casos, você precisa construir um validador ou filtro que empregue uma abordagem de lista branca.

7.5 Melhores práticas de segurança

Nos próximos cinco capítulos, vamos tratar de tópicos específicos de segurança. Mas isso não nos isenta de tentar apavorá-lo, quer dizer, conscientizá-lo da necessidade de gerar código seguro, indo além do que o Laminas provê.

Há três regras de ouro para seguir quando fornecer segurança para aplicações web e seus ambientes:

1. Sempre valide sua entrada.
2. Sempre filtre sua saída.
3. Nunca confie em seus usuários.

O desenvolvimento de código seguro é um grande desafio, pois não envolve somente o uso de técnicas, mas conscientização de toda a equipe de desenvolvimento. Para tentar diminuir (não eliminar) ao máximo a possibilidade de uma falha de segurança, deve-se adotar uma postura rígida, sem concessões.

Nesse ponto, cabe uma metáfora com as histórias em quadrinhos. O Superman e o Batman são companheiros de equipe (Liga da Justiça da América) e grandes amigos. O Superman é um símbolo de integridade e altruísmo, alguém cuja honestidade nunca é posta em dúvida. Mesmo assim, o Batman sempre carrega uma pedra de kriptonita no seu cinto de utilidades. Os membros da Liga dizem que o Batman é paranoico. Na verdade, o Batman está totalmente alinhado com princípios de segurança.

Batman conhece os quatro princípios enunciados por Howard e Leblanc (2005, p. 47-48) como *A vantagem do invasor e o dilema do defensor*:

1. O defensor deve defender todos os pontos; o invasor pode escolher o ponto mais fraco.
2. O defensor pode se defender somente de ataques conhecidos; o invasor pode investigar apenas vulnerabilidades desconhecidas.

3. O defensor deve estar constantemente vigilante; o invasor pode atacar a qualquer momento.
4. O defensor deve jogar de acordo com as regras; o invasor pode jogar sujo.

É claro que o Batman tem uma vantagem: ele opera fora do sistema. Você, infelizmente, não terá esse privilégio.

Segundo Howard e Leblanc (2005, p. 123), quando nossa aplicação enfrenta uma ameaça, podemos, em resposta, fazer o seguinte:

- Nada.
- Informar o usuário sobre a ameaça, exibindo, por exemplo, uma mensagem como "Houston, temos um problema".
- Remover o problema.
- Corrigir o problema.

Observe que remover o problema e corrigir o problema são coisas diferentes. Remover o problema é uma abordagem evasiva. Eu conheço um programador extremamente astuto que se gaba de produzir código sem erros. Ele simplesmente apaga os trechos de código que dão erro. Isso não resolve o problema, porque, além de provavelmente eliminar funcionalidade, não mostra o motivo de aquele trecho provocar erro.

Princípios de segurança

Conheço outro programador que não se surpreende com nada e parece ter uma resposta para tudo, mesmo que seja inventada. Esse programador se gaba de manter tudo em sua cabeça, menosprezando a documentação de processos e programas.

No livro *Um estudo em vermelho*, de Sir Arthur Conan Doyle, Sherlock Holmes afirma que a mente humana não é um quarto com paredes elásticas. E se algumas pessoas têm um sistema de fluxo de

trabalho implantado em sua mente, isso definitivamente não é algo comum.

O conhecimento não deve ficar na cabeça das pessoas. Isso pode funcionar para empresas de um homem só, mas não para uma organização de verdade. O conhecimento deve estar documentado dentro da empresa; deve haver métodos e padrões que todos seguem, de modo que um projeto de software não dependa das pessoas, mas das funções que elas exercem.

Isso pode parecer assustador para aqueles que querem garantir seu emprego escondendo informação, o que não é uma atitude profissional. Porém, devo lembrar de que o uso de métodos e padrões é uma atitude disciplinadora, e um programador disciplinado cria código com maior qualidade. Por isso, sempre é bom ter uma referência escrita de passos claros que você possa usar para verificar se está fazendo seu trabalho da forma como deveria ser feito. Listas de verificação são amigas.

Howard e Leblanc (2005, p. 77-90) enunciam os seguintes princípios para criar softwares seguros, que você pode usar para montar sua lista de verificação de código seguro:

- Aprenda com os erros (por isso eu não fujo dos erros em treinamentos, eu os exploro).
- Minimize sua área de ataque (frustre o atacante).
- Empregue *defaults* seguros (configuração padrão que minimize riscos) ou desative configurações padrão que possam ser exploradas maliciosamente.
- Utilize defesa em profundidade (use o Princípio da Baioneta – veja o box).
- "Imagine que seu aplicativo seja o último componente remanescente e que todos os mecanismos defensivos que o protegem foram destruídos. Neste momento, você deve proteger a si próprio." (HOWARD e LEBLANC, 2005)
- Utilize o menor privilégio (não é preciso poderes telecinéticos para realizar tarefas ordinárias).

- Compatibilidade com versões anteriores sempre lhe dará dores de cabeça (segurança exige mudança).
- Assuma que os sistemas externos são inseguros (não confie em ninguém).
- Planeje-se para falhas (elas acontecerão, com certeza).
- Em caso de falha, escape para um modo de segurança (o software precisa de um plano de contingência).
- Lembre-se de que recursos de segurança são diferentes de recursos seguros (os primeiros protegem, os segundos estão protegidos).
- Nunca baseie a segurança somente na obscuridade (esconder algo não o torna invulnerável).
- Não misture código e dados (ou operações e dados).
- Corrija as questões de segurança corretamente (não use XGH).

Princípio da Baioneta

Não dependa de um único recurso para se proteger. A baioneta é uma combinação de arma de fogo com arma branca. Quando não é possível atirar, ela pode ser usada como uma lança.

Técnicas de segurança

Princípios são bons porque dizem o que você deve fazer. Mas por si só não são suficientes. Não basta saber o que deve ser feito. É preciso saber como fazê-lo, se não você vira um hobbit rumo à Montanha Solitária para derrotar um dragão sem a mínima noção de como fará para concretizar seu intento.

Howard e Leblanc (2005, p. 77-90), sabiamente, enunciam técnicas de segurança que aplicadas em conjunto ajudam a seguir os princípios citados anteriormente. Destacamos algumas delas que tratamos neste livro:

- Autenticação (veremos como `Laminas\Authentication` implementa isso).

- Autorização (veremos como `Laminas\Permissions` implementa isso).
- Auditoria (veremos como `Laminas\Log` implementa isso).

Além disso, é importante utilizar criptografia para ofuscar conteúdo sensível, como senhas. PHP tem extensões para criptografia, como `Hash`, `OpenSSL` e `Sodium.

> **CUIDADO COM OS ALGORITMOS DE CRIPTOGRAFIA**
>
> Alguns dos primeiros algoritmos criptográficos, como `md4`, `md5` e `sha1`, demonstraram ser propensos a ataques de colisão e geralmente é recomendado não mais usá-los para aplicações criptográficas. O PHP tem funções que criptografam com os algoritmos `md5` e `sha1`, mas elas NUNCA devem ser usadas de forma isolada em produção. Você pode construir funções de criptografia que usem essas funções do PHP como parte do processo, mas não como a única transformação realizada.

Tenha consciência de que um atacante determinado não ficará satisfeito enquanto não causar um estrago. Um atacante de um sistema computacional é como uma criança travessa que não sossega enquanto não quebrar algo. Segundo Howard e Leblanc (2005, p. 279-292), atacantes podem tentar provocar o travamento do aplicativo ou do sistema operacional, ou ambos, esgotar a CPU, a memória, os recursos e até a largura de banda de rede.

O aplicativo sozinho não pode lidar com tudo isso. Na verdade, não é responsabilidade apenas dele. No entanto, ele pode tornar a tarefa do atacante mais difícil.

Não esgotamos o tema segurança neste capítulo. Conforme já foi dito, ele será aprofundado nos próximos capítulos.

CAPÍTULO 8
Filtros e conversores de dados

"(...) e lhe disse: até aqui virás, porém não mais adiante." – Jó 38:11a

Neste capítulo, você aprende que filtros não são apenas usados para fazer café ou beber água potável. O componente de filtragem `Laminas\Filter` do Laminas ajuda o desenvolvedor a não deixar conteúdo indesejável entrar ou sair da aplicação, além de convertê-lo para o formato que desejarmos.

8.1 Laminas\Filter

Filtros produzem algum subconjunto da entrada original, baseados em algum critério ou em uma transformação. Dentro de aplicações web, filtros são usados para remover entrada ilegal, espaços em branco desnecessários e executar outras funções. Uma transformação comum aplicada dentro de aplicações web é tratar entidades HTML. Um exemplo disso foi dado com o componente `Laminas\Escaper` no capítulo anterior.

`Laminas\Filter` provê um conjunto de filtros de dados geralmente necessários para aplicações web. Ele também fornece um mecanismo de encadeamento de filtro simples pelo qual múltiplos filtros podem ser aplicados para um dado simples em uma ordem definida pelo usuário.

`Laminas\Filter\FilterInterface` define a funcionalidade básica para o componente (converter um valor em outro) e requer um único método `filter()` a ser implementado por uma classe de filtro.

Aqui está um exemplo de um filtro que remove tags, neutralizando formatações e eventos indesejados:

```
<?php
use Laminas\Filter\StripTags;
require 'vendor/autoload.php';
$filter = new StripTags();
// Daredevil
echo $filter->filter('<font color="red">Daredevil</font>');
// while (true){window.alert("Oi como vai, tudo bem?");}
echo $filter->filter('<script type="text/javascript">while (true)
{window.alert("Oi como vai, tudo bem?");}</script>');
```

8.2 Filtros predefinidos

O componente laminas-filter oferece um conjunto de classes de filtro padrão com o framework. Para filtros que requerem identificação da localidade e do idioma, é necessário usar o componente laminas-i18n.

Classe	Descrição
`AllowList`	Retorna nulo se o valor não está presente em uma lista.
`Alnum`	Retorna só dados alfanuméricos de uma determinada localidade. Requer laminas-i18n.
`Alpha`	Retorna só dados alfabéticos de determinada localidade. Requer laminas-i18n.
`Basename`	Retorna só a parte do nome do arquivo de um caminho de diretório.
`Boolean`	Retorna o equivalente booleano em PHP do valor.
`Callback`	Retorna o valor transformado por uma função ou um método de classe.
`Compress`	Retorna textos, arquivos ou diretórios compactados.
`Decompress`	Retorna textos, arquivos ou diretórios descompactados.
`DenyList`	Retorna nulo se o valor está presente em uma lista.
`Digits`	Retorna só dígitos numéricos.
`Dir`	Retorna só o componente diretório de um caminho de diretório.
`HtmlEntities`	Converte caracteres "especiais" em entidades HTML.
`NumberFormat`	Converte um número para texto formatado de acordo com a localidade. Requer laminas-i18n.
`NumberParse`	Converte um texto para um número de acordo com a localidade. Requer laminas-i18n.

Classe	Descrição
`PregReplace`	Retorna um texto modificado a partir de uma expressão regular.
`RealPath`	Retorna o caminho absoluto, sem links simbólicos, `'/./'` e `'/../'`.
`StringPrefix`	Adiciona um prefixo
`StringSuffix`	Adiciona um sufixo
`StringToLower`	Retorna os caracteres convertidos para letras minúsculas.
`StringToUpper`	Retorna os caracteres convertidos para letras maiúsculas.
`StringTrim`	Retorna o texto sem espaços em branco à direita e à esquerda.
`StripNewLines`	Remove os caracteres que indicam uma nova linha (`\n\r`).
`StripTags`	Remove as tags HTML e PHP.
`ToFloat`	Retorna só números do tipo float.
`ToInt`	Retorna só números inteiros.
`ToNull`	Retorna NULL
`ToString`	Retorna string

Cada filtro recebe argumentos diversos em seu construtor, de acordo com a transformação que ele vai realizar. A seguir, descrevemos os argumentos dos construtores das classes de filtro predefinidas, quando existirem. Colchetes indicam argumentos opcionais. Quando mais de um tipo é possível para um argumento, usamos a barra em pé para indicar que pode ser um ou outro.

Classe	Descrição
`AllowList`	[$options]
`Alnum`	[boolean $allowWhiteSpace [, string $locale]]
`Alpha`	[boolean $allowWhiteSpace [, string $locale]]
`Boolean`	[$typeOrOptions, [$casting, [$translations]]]
`Callback`	[$callbackOrOptions[, $callbackParams]]
`Compress`	[array $options]
`Decompress`	[array $options]
`DenyList`	[$options]
`HtmlEntities`	[array $options]
`NumberFormat`	[string $locale [, int $style [, int $type]]]
`NumberParse`	[string $locale [, int $style [, int $type]]]
`PregReplace`	[array $options]
`RealPath`	[boolean $existsOrOptions]
`StringPrefix`	array $prefix
`StringSuffix`	array $suffix
`StringToLower`	[$encodingOrOptions]
`StringToUpper`	[$encodingOrOptions]
`StringTrim`	[$charlistOrOptions]
`StripTags`	[$options]

Você pode estar notando alguma semelhança dos nomes de alguns filtros com funções PHP. Não é mera coincidência. Os filtros encapsulam funções PHP, mas fazem mais do que apenas chamá-las.

Eles proveem uma estrutura para realizar conversões complexas em cadeia.

8.3 Cadeias de filtro

Quando usar múltiplos filtros, frequentemente será necessário impô-los em uma ordem específica, conhecida como "encadeamento". `Laminas\Filter\FilterChain` provê um modo simples de encadear filtros.

Aqui temos um exemplo de filtragem de um nome de usuário que é transformado em um texto com letras minúsculas e apenas com caracteres alfabéticos. Note que os caracteres não alfabéticos são removidos antes que quaisquer caracteres sejam convertidos para maiúsculas por causa da ordem dos filtros dentro do código.

```
<?php
use Laminas\Filter\StringToLower;
use Laminas\I18n\Filter\Alpha;
use Laminas\Filter\FilterChain;
// Cria uma cadeia de filtros e adiciona filtros à cadeia
$filterChain = new FilterChain();
$filterChain->attach(new Alpha())
->attach(new StringToLower());
// Filtra o nome de usuário
$username = $filterChain->filter($_GET['username']);
echo $username;
```

> **Nota**: Filtros são executados na ordem em que eles foram adicionados à cadeia.

8.4 Criando filtros customizados

Criar um filtro é um processo simples – uma classe precisa somente implementar `Laminas\Filter\FilterInterface` , que consiste de um único método `filter()` . Objetos cuja classe implemente essa interface podem ser adicionados a uma cadeia de filtro usando o método `attach()` de `Laminas\Filter\FilterChain` . Um exemplo de código é fornecido a seguir:

```
<?php
require_once 'vendor/autoload.php';
class MyFilter implements Laminas\Filter\FilterInterface {
    public function filter($value) {
    // executa transformação sobre $value para chegar a $valueFiltered
        return $valueFiltered;
    }
}
```

8.5 Laminas\InputFilter\InputFilter

`Laminas\InputFilter\InputFilter` fornece uma interface declarativa para associar múltiplos filtros e validadores, aplicá-los a coleções de dados e recuperar valores de entrada depois que eles forem processados pelos filtros e validadores. Os valores são retornados em um formato tratado por padrão para saída segura de HTML.

- Filtros transformam valores de entrada pela remoção ou alteração de caracteres dentro do valor. O objetivo é "normalizar" valores de entrada até que eles casem com um formato esperado.
- Validadores verificam valores de entrada contra critérios e reportam se eles passaram pelo teste ou não. O valor não é alterado, mas a verificação pode falhar.

O processo para usar `Laminas\InputFilter\InputFilter` :

1. Declare as regras de filtragem e validação.
2. Crie o processador de filtros e validadores (`Laminas\InputFilter\InputFilter`).
3. Associe os filtros e validadores ao processador.
4. Forneça os dados de entrada.
5. Recupere os campos, filtrados e validados, e as mensagens de erro.

A criação de um objeto associador de filtros e validadores, ou de um filtro de entrada, como sugere o nome da classe, é feita com a seguinte sintaxe:

```
$inputFilter = new Laminas\InputFilter\InputFilter();
```

Para associar uma cadeia de filtros ao filtro de entrada, use o método `setFilterChain()` :

```
$inputFilter->setFilterChain($filterChain);
```

Para associar uma cadeia de validadores ao filtro de entrada, use o método `setValidatorChain()` :

```
$inputFilter->setValidatorChain($validatorChain);
```

Você pode utilizar seus próprios filtros e validadores.

CAPÍTULO 9
Validadores de dados

Ignoro o que sejam princípios, a não ser que se tratem de regras que se prescrevem aos outros para nosso proveito. – Denis Diderot

Neste capítulo, aprendemos a responder à temível pergunta: será que os dados são válidos? Segundo Charles Babbage, um sistema de computação compõe-se de entrada, processamento e saída.

Porém, se o processamento for feito em cima de uma entrada inválida, a saída será, por consequência, inválida. E não podemos nos esquecer de que as decisões são tomadas em função da saída de sistemas de informação.

9.1 Laminas\Validator

Validadores examinam a entrada contra algum conjunto de requisitos e produzem um resultado booleano (`TRUE` ou `FALSE`), de acordo com o que foi passado. Eles podem fornecer mais informações sobre quais requisitos a entrada não atendeu no evento de falha de validação.

`Laminas\Validator` fornece um conjunto de validadores de necessidade mais comum para aplicações web. Ele também provê um mecanismo de encadeamento de validadores simples, pelo qual múltiplos validadores podem ser aplicados para um dado único, em uma ordem definida pelo usuário.

`Laminas\Validator\ValidatorInterface` define a funcionalidade básica para o componente, estabelecendo dois métodos essenciais.

Método	Descrição
`isValid()`	Executa validação sobre os valores de entrada, retornando true em caso de sucesso no atendimento aos critérios, ou FALSE , em caso de falha. No último caso, o próximo método é usado.
`getMessages()`	Retorna um array de mensagens explicando as razões para a falha de validação. As chaves de array identificam as razões com códigos string, e os valores do array fornecem mensagens string legíveis. Esses pares chave-valor são dependentes de classe, e cada classe também tem uma definição de constante que casa cada identificador com uma causa de falha de validação.

Note que cada chamada a `isValid()` sobrescreve a chamada precedente. Portanto, cada saída de `getMessages()` será aplicada somente à validação mais recente.

```
<?php
use Laminas\Validator\EmailAddress;
require_once 'vendor/autoload.php';
$email = 'emailinvalido';
$validator = new EmailAddress();
if ($validator->isValid($email)) {
// O e-mail parece ser válido
} else {
// O e-mail é inválido; imprime as razões
    foreach($validator->getMessages()as $messageId => $message) {
        echo "Falha de validação '$messageId': $message\n";
    }
}
```

9.2 Customizando mensagens

Classes de validação fornecem um método `setMessage()` com o qual você pode especificar o formato da mensagem retornada por `getMessages()` sobre a falha de validação.

O primeiro argumento contém uma string com a própria mensagem de erro. Tokens podem ser incorporados dentro da string para customizar a mensagem com dados relevantes. O token `%value%` é suportado por todos os validadores, enquanto outros tokens são suportados em uma base caso a caso, de acordo com a classe de validação.

O segundo argumento contém uma string com o template de falha de validação a ser configurado, usado quando uma classe de validação define mais do que uma causa para falha.

Se o segundo argumento estiver faltando, `setMessage()` assume que a mensagem deve ser usada para o primeiro template de mensagem declarado na classe de validação. Como muitas classes de validação têm somente um template de mensagem de erro definido, não há necessidade de especificar para qual template mudar.

Um exemplo de código é fornecido a seguir:

```
<?php
use Laminas\Validator\StringLength;
require_once 'vendor/autoload.php';
$validator = new StringLength(8);
$validator->setMessage(
'A frase \'%value%\' é muito curta; deve ter ao menos %min% caracteres',
StringLength::TOO_SHORT);
if (!$validator->isValid('word')) {
    $messages = $validator->getMessages();
    echo current($messages);
    // Imprime "A frase 'word' é muita curta; deve ter ao menos 8
caracteres"
}
```

9.3 Validadores predefinidos

O componente laminas-validator oferece um conjunto de classes padrão de validação, para verificações frequentes em aplicações web. Para validações que requeiram a identificação da localização e do idioma, você pode encontrar mais classes no componente laminas-i18n, que abordaremos com mais detalhes no capítulo 15.

Classe	Descrição
`Alnum`	Verifica se o valor contém somente caracteres alfabéticos e dígitos. Requer o componente laminas-i18n.
`Alpha`	Verifica se o valor contém somente caracteres alfabéticos. Requer o componente laminas-i18n.
`BackedEnumValue`	Verifica se o valor (string ou numérico) faz parte de um tipo `enum`.
`Barcode`	Verifica se um código de barras é válido de acordo com um determinado formato.
`Between`	Verifica se um valor se encontra entre um limite mínimo e máximo.
`Callback`	Verifica se um valor é válido segundo uma função ou um método especificado.
`Conditional`	Executa uma ou mais validações se uma condição especificada for atendida.
`CountryCode`	Verifica se um valor é um código de país válido segundo a ISO-3166

Classe	Descrição
	allpha 2. Requer o componente laminas-i18n.
`CreditCard`	Verifica se um valor é um número de cartão de crédito válido.
`Date`	Verifica se o valor é uma data válida (a localização é configurável).
`DateTime`	Verifica se o valor é uma data, hora ou data e hora válida (a localização é configurável). Requer o componente laminas-i18n.
`DateComparison`	Verifica se uma data encontra-se em um determinado intervalo
`DateIntervalString`	Verifica se valor é uma especificação DateInterval válida
`Digits`	Verifica se o valor contém somente dígitos.
`EmailAddress`	Verifica se o valor é um endereço de e-mail válido.
`EnumCase`	Verifica se uma string é um caso válido de um tipo `enum`
`Explode`	Executa um validador para cada item de um array gerado pela explosão de uma string
`Hex`	Verifica se o valor contém apenas dígitos hexadecimais.
`Hostname`	Verifica se o valor é um DNS, endereço IP ou localhost.

Classe	Descrição
`HostWithPublicIPv4Address`	Verifica se um endereço IP ou hostname não é reservado
`Iban`	Verifica se um valor é um número IBAN (International Bank Account Number).
`Identical`	Verifica se um valor é idêntico a um token.
`InArray`	Verifica se um valor está contido em um array .
`Ip`	Verifica se o valor é um endereço IP válido.
`IsArray`	Verifica se o valor é um array.
`Isbn`	Verifica se o valor é um ISBN-10 ou ISBN-13.
`IsCountable`	Verifica se o valor é contável (array ou objeto que implementa Countable)
`IsInstanceOf`	Verifica se o valor é uma instância de uma classe ou interface
`IsInt`	Verifica se o valor é um número inteiro. Requer o componente laminas-i18n.
`IsFloat`	Verifica se o valor é ponto flutuante. Requer o componente laminas-i18n.
`NotEmpty`	Verifica se o valor não é vazio.
`PhoneNumber`	Verifica se o valor é um número de telefone válido. Requer o componente laminas-i18n.

Classe	Descrição
`PostCode`	Verifica se o valor é um código postal válido de acordo com a localidade. Requer o componente laminas-i18n.
`Regex`	Verifica se o valor casa com uma expressão regular.
`Step`	Verifica se o valor é um elemento válido de uma progressão aritmética.
`StringLength`	Verifica se o comprimento do valor como texto está dentro de um limite mínimo e máximo.
`Timezone`	Verifica se um valor string é um `timezone` (fuso horário) válido
`Uri`	Verifica se um valor string é um URI válido
`Uuid`	Verifica se o valor é um UUID válido

Cada validador recebe argumentos diversos em seu construtor, de acordo com a transformação que ele for realizar. A seguir, descrevemos os argumentos dos construtores das classes de validação mencionadas. Colchetes indicam argumentos opcionais. Quando mais de um tipo é possível para um argumento, usamos a barra em pé para indicar que pode ser um ou outro.

Classe	Construtor
`Alnum`	[boolean $allowWhiteSpace]
`Alpha`	[boolean $allowWhiteSpace]
`BackEnumValue`	array $options
`Barcode`	[array $options]
`Between`	[array $options]
`Callback`	[array $options]
`Conditional`	ValidatorChainFactory $chainFactory, array $options
`CountryCode`	[array $options]
`CreditCard`	[array $options]
`Date`	[array $options]
`Date`	[$options]
`DateComparison`	[array $options]
`DateIntervalString`	[array $options]
`Digits`	[array $options]
`EmailAddress`	[array $options]
`EnumCase`	array $options
`Explode`	[array $options]
`Hex`	[array $options]
`Hostname`	integer $allow[,bool $validateIdn[,bool $validateTld]]
`HostWithPublicIPv4Address`	[array $options]
`Iban`	[array $options]

Classe	Construtor
`Identical`	[mixed $token]
`InArray`	[array $options]
`Ip`	[array $options]
`IsArray`	[array $options]
`Isbn`	[array $options]
`IsCountable`	[array $options]
`IsInstanceOf`	array $options
`IsInt`	[$options]
`IsFloat`	[$options]
`NotEmpty`	[array $options]
`PhoneNumber`	[$options]
`PostCode`	[$options]
`Regex`	[string|array $options]
`Step`	[array $options]
`StringLength`	[array $options]
`Timezone`	[array $options]
`Uri`	[array $options]
`Uuid`	[array $options]

9.4 Cadeias de validação

Quando usar múltiplas validações, será frequentemente necessário impô-las em uma ordem específica, conhecida como "encadeamento". `Laminas\Validator\ValidatorChain` fornece um modo simples de encadeá-las. Validadores são executados na ordem em que foram adicionados a `Laminas\Validator\ValidatorChain`.

Exemplo: validar se o nome de usuário está entre 6 e 12 caracteres alfanuméricos.

```
<?php
use Laminas\I18n\Validator\Alnum;
use Laminas\Validator\StringLength;
use Laminas\Validator\ValidatorChain;
require_once 'vendor/autoload.php';
$username = '1234cebolinhasimbaursinhopooh!@#$%*()';
// Cria uma cadeia de validação e adiciona validadores para isso
$validatorChain = new ValidatorChain();
$validatorChain
->addValidator(new StringLength(6,12))
->addValidator(new Alnum());
// Valida o nome de usuário
if ($validatorChain->isValid($username)) {
// O nome de usuário passou pela validação
} else {
// O nome de usuário falhou na validação; imprime as razões
    foreach ($validatorChain->getMessages() as $message) {
        echo "$message\n";
    }
}
```

9.5 Criando validadores customizados

Além dos validadores padrões fornecidos pelo Laminas, `Laminas\Validator\ValidatorInterface` define dois métodos, `isValid()` e `getMessages()`, que podem ser implementados por classes de usuário para criar objetos de validação customizados. Esses objetos podem

ser adicionados a uma cadeia de validação usando `Laminas\Validate\ValidatorChain`, ou com `Laminas\InputFilter\InputFilter`.

Como mencionado anteriormente, resultados de validação são retornados como valores booleanos, possivelmente junto a informações, como "por que a validação falhou" (útil para análise de usabilidade).

`Laminas\Validator\AbstractValidator` é usado para implementar a funcionalidade básica de mensagem de falha de validação, que pode ser estendida. A classe filha utilizaria a lógica do método `isValid()` e definiria variáveis de mensagem e templates de mensagem que correspondem aos tipos de falha de validação que podem ocorrer.

Geralmente, o método `isValid()` não lançará quaisquer exceções, a menos que seja impossível determinar se o valor de entrada é válido ou não (exemplos: um servidor LDAP não pode ser contatado, um arquivo não pode ser aberto, um banco de dados está indisponível etc.)

Capítulo 10
Autenticação

O autêntico, o verdadeiro grande talento descobre as suas maiores alegrias na realização. – Johann Wolfgang Von Goethe

Neste capítulo você aprenderá a identificar os usuários de sua aplicação, construindo uma muralha em torno dela com apenas uma passagem: a autenticação. Autenticação é o processo de verificar a identidade de um usuário contra algum conjunto de critérios preestabelecidos. É como responder à pergunta: "Eles são quem afirmam ser?".

`Laminas\Authentication` fornece uma API para conduzir a autenticação, junto com adaptadores projetados para a maioria dos usos comuns. Os adaptadores autenticam contra um serviço particular, como LDAP, RDBMS etc.

Enquanto seu comportamento e suas opções variam, eles compartilham ações comuns:

- aceitam credenciais de autenticação;
- executam consultas contra o serviço;
- retornam resultados.

10.1 Laminas\Authentication\ AuthenticationService

Este componente fornece uma API para autenticação e inclui adaptadores para os cenários de uso mais comuns. Para realizar

uma autenticação, instancie `AuthenticationService` e injete um adaptador, conforme exemplo a seguir.

```
$authentication = new AuthenticationService();
$authentication->setAdapter($adapter);
```

10.2 Persistência de identidade

Um aspecto importante do processo de autenticação é a habilidade de reter a identidade e persisti-la ao longo das requisições de acordo com a configuração de sessão do PHP. A classe de armazenamento padrão no Laminas é `Laminas\Authentication\Storage\Session` (que usa `Laminas\Session`).

Uma classe customizada pode ser usada em seu lugar desde que ela implemente:

`Laminas\Authentication\Storage\StorageInterface`

e, por consequência, seu método `setStorage()`. A persistência pode ser customizada pelo uso de uma classe adaptadora de forma direta, dispensando o uso de `Laminas\Authentication\AuthenticationService`.

A identidade é persistida por meio do método `write()` do objeto armazenador. Para obter a instância do objeto armazenador, usamos o método `getStorage()` do autenticador. Desta forma, para persistir dados de identidade, usamos uma chamada assim:

```
$authentication->getStorage()->write($identity);
```

10.3 Resultados de autenticação

`Laminas\Authentication\AuthenticationService` possui adaptadores que retornam uma instância de `Laminas\Authentication\Result` com o método `authenticate()` para representar os resultados de uma tentativa de autenticação.

Os quatro métodos da classe `Result`, exibidos a seguir, podem fornecer um conjunto comum de operações para manipular os resultados:

Método	Descrição
`isValid()`	Retorna `true` se, e somente se, o resultado representa uma tentativa de autenticação que teve sucesso.
`getCode()`	Retorna um identificador constante de `Laminas\Authentication\Result` para confirmar o sucesso ou determinar o tipo de falha de autenticação.
`getIdentity()`	Retorna a identidade da tentativa de autenticação (que pode ser persistida).
`getMessages()`	Retorna um array de mensagens reportando uma tentativa frustrada de autenticação.

Os adaptadores `Laminas\Authentication` permitem o uso de diversas tecnologias de autenticação, tais como:

- Digest – `Laminas\Authentication\Adapter\Digest`
- LDAP – `Laminas\Authentication\Adapter\Ldap`
- HTTP – `Laminas\Authentication\Adapter\Http`
- Tabela de banco de dados – `Laminas\Authentication\Adapter\DbTable\CredentialTreatmentAdapter` ou `Laminas\Authentication\Adapter\DbTable\CallbackCheckAdapter` (para validação adicional)

10.4 Retornos possíveis para uma tentativa de autenticação

Conforme apresentado no quadro de métodos da classe `Laminas\Authentication\Result`, o método `getCode()` retorna um identificador de sucesso ou falha da tentativa de autenticação. Esse identificador é um número inteiro. Mas para que usar números? Use constantes! `Laminas\Authentication\Result` define as seguintes:

Constante	Descrição
`FAILURE`	Falha geral
`FAILURE_IDENTITY_NOT_FOUND`	A identidade não foi encontrada no repositório
`FAILURE_IDENTITY_AMBIGUOUS`	Há mais de uma identidade no meio de armazenamento
`FAILURE_CREDENTIAL_INVALID`	A credencial é inválida. Credencial, neste caso, é um nome mais bonito para senha
`FAILURE_UNCATEGORIZED`	A razão da falha não se encaixa em nenhuma categoria
`SUCCESS`	A tentativa de autenticação foi um sucesso!

10.5 Criação de adaptadores customizados de autenticação

Você quer criar uma classe para realizar autenticação contra uma tecnologia para a qual o Laminas ainda não tem uma classe? Sem problemas. Sua classe tem de estender `Laminas\Authentication\Adapter\AbstractAdapter`.

O exemplo a seguir ilustra uma autenticação contra um adaptador indeterminado, mostrando que a fonte de dados contra o qual se autentica é indiferente para o componente `Laminas\Authentication`:

```
$authentication = new AuthenticationService();
// injeta o adaptador no serviço de autenticação
$authentication->setAdapter($adapter);
// autentica
$result = $authentication->authenticate();
// verifica o resultado da autenticação
if ($result->isValid()) {
    // obtém a identidade do usuário
    $identity = $result->getIdentity();
    // persiste a identidade
    $authentication->getStorage()->write($identity);
}
else {
    // obtém as mensagens de erro
    $messages = $result->getMessages();
}
```

Como eu sei se há uma identidade armazenada?

Use o método `hasIdentity()` de `Laminas\Authentication\AuthenticationService`. Ele diz se tem uma identidade. Se quiser recuperá-la, use `getIdentity()`, como visto anteriormente. O exemplo a seguir mostra o esquema geral da verificação de identidade.

```
$authentication = new AuthenticationService();
if ($authentication->hasIdentity()) {
    $identity = $authentication->getIdentity();
}
```

10.6 Remoção da identidade armazenada

Em algum momento será necessário desconectar o usuário da aplicação e isso implica na remoção de seus dados de identidade. Para fazer isso, use o método `clearIdentity()` de `Laminas\Authentication\AuthenticationService`.

10.7 Implementando autenticação na aplicação

Copie o projeto corps2 como corps3. Vamos implementar a autenticação no novo projeto usando o banco de dados como provedor.

Tabela de usuários

A primeira providência é criar uma tabela para os usuários no banco de dados `corps`. A declaração SQL para criar essa tabela, pelo menos no MySQL, é a seguinte:

```
CREATE TABLE usuarios(uid integer AUTO_INCREMENT, identidade varchar(30),
credencial varchar(255), PRIMARY KEY (uid));
```

O script completo para criação da tabela de usuários, em uma forma alternativa, está disponível em https://github.com/fgsl/phpbooksample/blob/main/sql/usuarios.sql

Formulário de autenticação

Crie a pasta `Form` dentro de `module\Application\src\Application`. Dentro da pasta criada, crie a classe `Login`, contendo o seguinte código-fonte:

```
<?php
namespace Application\Form;
```

```
use Fgsl\Form\AbstractForm;
class Login extends AbstractForm
{
    public function __construct($name = null) {
        parent::__construct('login');
        $this->setAttribute('method', 'post');
        $this->addElement('identidade', self::TEXT, 'Usuário',
['autofocus'=>'autofocus']
        )
        ->addElement('credencial', self::PASSWORD, 'Senha')
        ->addElement('submit', self::SUBMIT, 'Entrar');
    }
}
```

Note que estamos usando uma constante `PASSWORD` da classe `Fgsl\Form\AbstractForm`. Essa classe faz parte de uma extensão do Laminas, o framework Fgsl. Você pode instalar essa extensão usando o Composer. Execute o seguinte comando na raiz do projeto:

```
composer require fgslframework/framework
```

Este comando perguntará se você deseja injetar a configuração do componente `Laminas\Paginator` em algum lugar assim:

```
Please select which config file you wish to inject 'Laminas\Paginator'
into:
```

Escolha a opção `[1] config/modules.config.php`. Em seguida, o comando perguntará se essa opção de injeção deve ser lembrada para outros pacotes:

```
Remember this option for other packages of the same type? (Y/n)`
```

Digite Y (yes) para evitar essa pergunta nas próximas vezes.

O comando `composer require` vai baixar o framework Fgsl e alterar o arquivo `composer.json`, além de alterar os arquivos responsáveis pelo autocarregamento de classes. O framework Fgsl contém várias extensões para componentes do Laminas, além do formulário, como

banco de dados, filtro de entrada de dados, MVC e injeção de dependências.

Modelo do usuário

Criaremos agora a classe que representará o usuário da aplicação, que será autenticado (ou não). Para isso, crie a pasta `Model` dentro de `module\Application\src\Application`, pois a classe a ser criada é um modelo. Dentro da pasta criada, crie a classe `Usuario`, de acordo com a listagem a seguir:

```
<?php
namespace Application\Model;
use Laminas\ServiceManager\ServiceManager;
use Laminas\Authentication\Adapter\DbTable;
use Laminas\Authentication\AuthenticationService;
class Usuario
{
    private string $identidade;
    private string $credencial;
    public array $messages = array();

    public function __construct(string $identidade, string $credencial) {
        $this->identidade = $identidade;
        $this->credencial = $credencial;
    }

    public function authenticate(ServiceManager $sm) {
    // cria o adaptador para o mecanismo contra o qual se fará a
autenticação
        $dbAdapter = $sm->get('LaminasDbAdapter');
        $tableAdapter = new DbTable($dbAdapter);
        $tableAdapter->setIdentityColumn('identidade')
            ->setTableName('usuarios')
            ->setCredentialColumn('credencial')
            ->setIdentity($this->identidade)
            ->setCredential($this->credencial);
        // cria o serviço de autenticação e injeta o adaptador nele
        $authentication = new AuthenticationService();
```

```
        $authentication->setAdapter($tableAdapter);
        // autentica
        $result = $authentication->authenticate();
        if ($result->isValid()) {
            // recupera o registro do usuário como um objeto, sem o campo
senha
            $usuario = $authentication->getAdapter()-
>getResultRowObject(null,'senha');
            $authentication->getStorage()->write($usuario);
            return true;
        }
        else {
            $this->messages = $result->getMessages();
            return false;
        }
    }
}
```

Ação para exibir o formulário e mensagens de erro

Para que o usuário se autentique, é necessária uma interface, que neste caso será um formulário com campos para digitar a identidade do usuário e sua credencial e que também avise os motivos da falha no processo de autenticação, caso ocorra. Para isso, vamos alterar o método `indexAction()` de `IndexController` no módulo `Application`, para que instancie a classe `Login`, configure a `action` do formulário para invocar o método `loginAction()` e recupere mensagens de erro geradas pelo processo de autenticação. O método ficará assim:

```
public function indexAction()
{
    $form = new Login();
    $form->setAttribute('action', $this->url()->fromRoute('application',
['controller'=>'index', 'action'=>'login']));
    $messages = '';
    if ($this->flashMessenger()->getMessages()) {
        $messages = implode(',', $this->flashMessenger()->getMessages());
    }
```

```
    return ['form'=>$form, 'messages'=>$messages];
}
```

Adicione o namespace da classe de formulário:

```
use Application\Form\Login;
```

Ação para autenticar o usuário

Vamos implementar o método `loginAction()` em `IndexController` para fazer a autenticação usando o modelo `Usuario`. O método ficará assim:

```
public function loginAction() {
    $request = $this->getRequest();
    $identidade = $request->getPost('identidade');
    $credencial = $request->getPost('credencial');
    $usuario = new Usuario($identidade, $credencial);
    if ($usuario->authenticate($this->sm)) {
        return $this->redirect()->toRoute('application',
['controller'=>'index','action'=>'menu']);
    }
    $this->flashMessenger()->addMessage(implode(',',$usuario->messages));
    return $this->redirect()->toRoute('home');
}
```

Adicione o namespace da classe `Model`:

```
use Application\Model\Usuario;
```

O atributo `$this->sm` é uma instância da classe `ServiceManager`. Esse atributo precisa ser declarado na classe `IndexController` assim:

```
private ?ServiceManager $sm = null;
```

Esse atributo requer a importação do seguinte namespace:

```
use Laminas\ServiceManager\ServiceManager;
```

Esse atributo precisa ser injetado no construtor da classe, que deve ser implementado assim:

```
public function __construct(ServiceManager $sm)
{
    $this->sm = $sm;
}
```

Quem injetará a instância de `ServiceManager` será uma classe fábrica (que implementa o padrão de projeto `Factory Method`). Crie no diretório `controller` a classe `IndexControllerFactory` com o seguinte código-fonte:

```
<?php
namespace Application\Controller;

use Laminas\ServiceManager\Factory\FactoryInterface;
use Psr\Container\ContainerInterface;

class IndexControllerFactory implements FactoryInterface
{

    public function __invoke(ContainerInterface $container,
$requestedName, ?array $options = null): IndexController
    {
        return new IndexController($container);
    }
}
```

Observe que o método `__invoke`, implementado por força do contrato com `FactoryInterface`, recebe como argumento um objeto da interface `ContainerInterface`. Esse objeto é uma instância de `ServiceManager`. Assim, a fábrica precisa de apenas uma linha para injetar o objeto no controlador. Mas a fábrica não é invocada automaticamente. Temos que dizer que ela deve ser usada no arquivo de configuração do módulo.

No arquivo `module.config.php` do módulo `Application`, há uma instrução que define a fábrica padrão de `IndexController`:

```
'factories' => [
    Controller\IndexController::class => InvokableFactory::class,
],
```

A classe `InvokableFactory` é uma fábrica padrão do Laminas, que apenas cria o objeto da classe indicada usando o operador `new`, sem passar nenhum argumento. Vamos substituir a classe `InvokableFactory` por `IndexControllerFactory` de modo que a instrução fique assim:

```
'factories' => [
    Controller\IndexController::class =>
Controller\IndexControllerFactory::class,
],
```

Ação para exibir o menu

Se o usuário tiver uma identidade e credencial válidos, ele deverá ser direcionado para uma página restrita para usuários autenticados, que neste caso será o menu principal da aplicação. Para isso, vamos implementar o método `menuAction()` de `IndexController` assim:

```
public function menuAction() {
    $authentication = new AuthenticationService();
    return ['usuario'=>$authentication->getIdentity()];
}
```

Adicione o namespace da classe autenticadora:

```
use Laminas\Authentication\AuthenticationService;
```

Trait para retornar ao menu

Vamos fazer com que as telas dos cadastros de setores espaciais e Lanternas Verdes exibam um hyperlink de retorno para a tela do menu principal. Para não sobrecarregar o método `indexAction()` nos dois controladores (`SetorController` e `LanternaController`), vamos usar um *trait* para permitir o compartilhamento dessa sobrecarga.

Poderíamos colocar o método a ser usado na classe abstrata e herdá-lo, mas podemos usar um *trait* para isolar e compartilhar o método. A vantagem do de um *trait* é que ele pode ser usado por classes diferentes sem a necessidade de herança entre elas, o que poderia compartilhar métodos não desejados.

Na pasta `Tropa\Controller` crie o arquivo `MenuTrait.php`. O Eclipse tem uma opção para criar um *trait* em `File → New → Trait`. O conteúdo do arquivo será o seguinte:

```
<?php
namespace Tropa\Controller;
trait MenuTrait
{
    public function indexAction()
    {
        $viewModel = parent::indexAction();
        $viewModel->setVariable('urlHomepage', $this->url()-
>fromRoute('application',
            ['controller'=>'index','action'=>'menu']
        ));
        return $viewModel;
    }
}
```

Nas classes `SetorController` e `LanternaController`, no mesmo módulo `Tropa`, adicione a seguinte linha, dentro do corpo da classe e antes de qualquer atributo:

```
use MenuTrait;
```

O código do *trait* será compartilhado pelas duas classes, fazendo com que seja possível navegar das telas de listagem dos cadastros para o menu principal.

Visão do formulário de autenticação

Precisamos exibir o formulário de autenticação que definimos anteriormente. Para fazer isso, copie o arquivo `index.phtml` de

`view\application\index` no módulo `Application` como `menu.phtml`. Em seguida, edite o arquivo `index.phtml` e altere seu conteúdo para o seguinte:

```
<?php
$title = 'Acesso ao Sistema';
$this->headTitle($title);
?>
<h1><?=$this->escapeHtml($title)?></h1>
<p><?=$this->messages?></p>
<?php
$form = $this->form;
$form->prepare();
echo $this->form()->openTag($form);
echo $this->formCollection($form);
echo $this->form()->closeTag();
?>
```

Exibindo a identidade do usuário no menu

Vamos editar o arquivo `menu.phtml`, do módulo `Application`, para que exiba a identidade do usuário. Altere-o de modo que fique como mostra a próxima listagem.

```
<?php
$title = 'Cadastros';
$this->headTitle($title);
?>
<h1><?=$this->escapeHtml($title)?></h1>
<p>Usuário: <?=$this->usuario->identidade?></p>
<?=$this->navigation('navigation')->menu()?>
```

Observe que temos um método `navigation()` sendo chamado logo após a exibição da identidade do usuário. Esse método criará um menu de acordo com dados de configuração que mostraremos na próxima seção.

Ação para desconectar o usuário

Vamos implementar o método `logoutAction()` de `IndexController`, no módulo `Application`, para que o usuário possa sair do sistema:

```
public function logoutAction() {
    $authentication = new AuthenticationService();
    $authentication->clearIdentity();
    return $this->redirect()->toRoute('home');
}
```

Agora precisamos alterar a configuração do menu de navegação para incluir um link que aponte para o método `logoutAction()`. Para isso precisamos criar a chave `navigation` no arquivo `global.php` da pasta `config\autoload`.

```
'navigation' => [
    'default' => [
        [
            'label' => 'Setores Espaciais',
            'route' => 'tropa',
            'controller'=> 'setor',
            'pages' => [
                [
                    'label' => 'Incluir setor',
                    'route' => 'tropa',
                    'controller'=> 'setor',
                    'action'=> 'edit'
                ]
            ]
        ],
        [
            'label' => 'Lanternas Verdes',
            'route' => 'tropa',
            'controller' => 'lanterna',
            'pages' => [
                [
                    'label' => 'Incluir Lanterna',
                    'route' => 'tropa',
                    'controller'=>'lanterna',
                    'action'=> 'edit'
```

```
                ]
            ]
        ],
        [
            'label' => 'Sair',
            'route' => 'application',
            'controller' => 'index',
            'action'=> 'logout'
        ]
    ]
],
```

Verificando a identidade

Utilizaremos `Laminas\EventManager` para fazer a verificação da identidade baseada no evento de roteamento. Isso garantirá que os cadastros só possam ser acessados se o usuário tiver passado pela autenticação. A verificação da identidade será uma regra de negócio encapsulada no modelo de usuário. Acrescente o método `verifyIdentity()` à classe `Application\Model\Usuario`, conforme a listagem a seguir:

```
<?php
public static function verifyIdentity(MvcEvent $e) {
    $application = $e->getApplication();
    $request = $application->getRequest();
    $uri = $request->getRequestUri();
    $baseUrl = $request->getBaseUrl();
    // separamos os segmentos do URL e recuperamos o último
    $segment = explode('/',$uri);
    $uri = end($segment);
    $baseUrl = str_replace('/','', $request->getBaseUrl());
    $uri = empty($uri) ? $baseUrl : $uri;
    if ($uri == $baseUrl || $uri == 'login'
|| $uri == 'application')
        return;
    $authentication = new AuthenticationService();
    if (!$authentication->hasIdentity()) {
    // preparamos um redirecionamento para a página de acesso ao sistema
```

```
        $router = $application->getServiceManager()->get('Router');
        $url = $router->assemble ( [
            'controller' => 'index',
            'action' => 'index'
        ], [
            'name' => 'application'
        ]
        );
        $response = $e->getResponse();
        $response->setHeaders($response->getHeaders()-
>addHeaderLine('Location',$url));
        $response->setStatusCode(302);
        $response->sendHeaders();
        exit ();
    }
}
```

O uso da classe `MvcEvent` requer a importação de um namespace na classe `Application\Model\Usuario`:

```
use Laminas\Mvc\MvcEvent;
```

Monitorando o evento de roteamento

A identidade do usuário deverá ser verificada antes que o controlador seja instanciado. A instanciação dos controladores ocorre após o roteamento, quando os URIs são casados com as rotas definidas. Por isso, vamos incluir o método `verifyIdentity()` de `Application\Model\Usuario` como observador do evento `Mvc::EVENT_ROUTE`. Faremos isso no método `onBootstrap()` da classe `Module` do módulo `Application`. Ele ficará assim:

```
public function onBootstrap(MvcEvent $e)
{
    $eventManager = $e->getApplication()->getEventManager();
    $defaultListener = new DefaultListenerAggregate();
    $defaultListener->attach($eventManager);
```

```
    $sessionManager = new SessionManager();
    $sessionManager->start();
    $eventManager->attach(MvcEvent::EVENT_ROUTE,
['Application\Model\Usuario','verifyIdentity']);
}
```

O uso da classe `MvcEvent` requer a importação de um namespace na classe `Module`:

```
use Laminas\Mvc\MvcEvent;
```

O uso da classe `DefaultListenerAggregate` requer a importação de outro namespace na classe `Module`:

```
use Laminas\ModuleManager\Listener\DefaultListenerAggregate;
```

O uso da classe `SessionManager` requer a importação de mais outro namespace na classe `Module`:

```
use Laminas\Session\SessionManager;
```

Você pode comprovar imediatamente a efetividade da verificação de identidade sem ter nenhum usuário cadastrado. Tente acessar os URLs dos cadastros:

- http://localhost:8000/tropa/lanterna
- http://localhost:8000/tropa/setor

Nos dois casos você será redirecionado para a página de acesso ao sistema, similar à da figura a seguir.

Figura 10.1: Página de login de corps3

Em seguida, inclua um usuário (diretamente no banco de dados) e autentique-se. Desse modo poderá constatar que somente autenticado poderá navegar pela aplicação. Com um usuário e senha válidos, após fazer o login, você deverá ver uma página assim:

Figura 10.2: Página do menu inicial de corps3

Observe que o menu configurado pelo componente `Laminas\Navigation` no arquivo `global.php` é apresentado de acordo

com a hierarquia definida pelo array de configuração.

Capítulo 11
Controle de permissões

Todo acesso a uma alta função se serve de uma escada tortuosa. – Francis Bacon

Este capítulo trata de controle de acesso ou de permissões, que é o sistema que garante que um determinado perfil de usuário só possa executar ações que foram explicitamente atribuídas a ele.

Um termo mais conciso para controle de permissões é autorização. Autorização é o processo de associar direitos para o usuário baseado em sua identidade. Algo como dizer "eles têm o direito de fazer as seguintes ações porque eles são quem eles são".

ACL é o acrônimo de *Access Control List* – Lista de Controle de Acesso. Uma ACL é como a lista de convidados de uma festa. Se você não está na lista, é um penetra e os seguranças altos e largos vão jogá-lo do outro lado da rua. Essa é a ideia que nossa aplicação tem de implementar para impedir que usuários realizem ações que não devem.

Laminas provê duas formas de controle de permissões, de modo que você pode escolher a implementação mais adequada ao modelo de negócio e política de segurança de sua aplicação.

11.1 Laminas\Permissions\Acl

Uma lista de controle de acesso define quem pode fazer o que em um sistema. É como um molho de chaves personalizado para cada funcionário de uma empresa. Ele só poderá entrar nas salas cujas chaves tiver.

`Laminas\Permissions\Acl` oferece uma funcionalidade de lista de controle de acesso enxuta e flexível e pode ser facilmente integrado com os componentes MVC do Laminas por meio de `Laminas\EventManager`, que já foi visto no capítulo sobre desenvolvimento orientado a eventos.

Tenha em mente que o componente `Laminas\Permissions\Acl` está envolvido somente com acesso de autorização, e, de qualquer maneira, não faz verificação de identidade (que é o processo de autenticação, executado no Laminas com `Laminas\Authentication`).

Pelo uso de `Laminas\Permissions\Acl`, uma aplicação controla como objetos de requisição (papéis) têm acesso garantido a objetos protegidos (recursos). Regras podem ser associadas de acordo com um conjunto de critérios – veja adiante na seção "Associando regras via asserções". A combinação dos processos de autenticação e autorização é popularmente chamada de "controle de acesso".

As regras de controle de acesso são especificadas com os métodos `allow()` e `deny()` de `Laminas\Permissions\Acl\Acl`. A verificação de acesso é feita com o método `isAllowed()`.

> **Nota** Chamar o método `isAllowed()` de `Laminas\Permissions\Acl\Acl` contra um papel ou recurso que não foi previamente adicionado à lista de controle de acesso resulta em uma exceção.

Algumas definições são necessárias antes de seguirmos em frente:

Quando falarmos em recurso no contexto de controle de acesso, estamos nos referindo a um objeto com acesso controlado. E, quando falamos em papel, estamos nos referindo a um objeto que solicita acesso a um recurso.

Não confunda papel com grupo ou usuário. Um grupo ou um usuário pode exercer um ou mais papéis, e um papel pode ser exercido por

um ou mais usuários e grupos. `Laminas\Permissions\Acl` não implementa usuários nem grupos, apenas papéis.

Em suma, papéis requisitam acesso a recursos, ou, mais especificamente, papéis solicitam autorização a recursos.

Temos uma última definição, que é o **privilégio**. O privilégio refere-se a uma ação específica que pode ser realizada com um recurso. Para alguns recursos existe somente um privilégio, que é o mero acesso. Porém, para outros, existem vários. Um exemplo seria o recurso documento, que pode prever as operações de criação, leitura, edição e remoção.

`Laminas\Permission\Acl` permite a associação de regras que afetam todos os privilégios de uma só vez ou privilégios específicos sobre um ou mais recursos.

Criando uma ACL

Mostraremos agora como criar uma ACL com `Laminas\Permissions\Acl`. No trecho de código a seguir, instanciamos uma lista de controle de acesso. Esse código deve anteceder o uso de qualquer recurso do sistema, idealmente após a autenticação do usuário.

```
<?php
use Laminas\Permissions\Acl\Acl;
$acl = new Acl();
```

Criando recursos

Uma classe precisa somente implementar `Laminas\Permissions\Acl\Resource\ResourceInterface`, que consiste do método único `getResourceId()`, para que `Laminas\Permissions\Acl` considere o objeto como um recurso. `Laminas\Permissions\Acl\Resource\GenericResource` é fornecida como uma implementação básica para extensibilidade. No trecho de código a

seguir criamos dois recursos, `'documento'` e `'impressora'`. Os nomes dos recursos referem-se ao que pretendemos controlar. Um recurso chamado `'documento'` é um nome conveniente para controlar o acesso a um arquivo de texto ou a uma planilha de custos, por exemplo. Um recurso chamado `'impressora'`, por sua vez, é um nome conveniente para controlar o acesso de usuários para um serviço de impressão.

```
<?php
use Laminas\Permissions\Acl\Resource\GenericResource;
$resource1 = new GenericResource ('documento');
$resource2 = new GenericResource ('impressora');
```

Criando papéis

Em um processo similar à criação de recursos, uma classe somente tem de implementar `Laminas\Permissions\Acl\Role\RoleInterface`, a qual consiste de um único método, `getRoleId()`, para que `Laminas\Permissions\Acl` considere o objeto como um papel. `Laminas\Permissions\Acl\Role\GenericRole` é fornecida como uma implementação básica para extensibilidade.

No trecho de código a seguir temos um exemplo de criação de dois papéis, `'chefe'` e `'peao'`. Em uma lista de controle de acesso as permissões são concedidas aos papéis e cada papel pode ter permissões diferentes. Assim, o `'chefe'` pode ter permissões diferentes do `'peao'`.

```
<?php
use Laminas\Permissions\Acl\Role\GenericRole;
$role1 = new GenericRole ('chefe');
$role2 = new GenericRole ('peao');
```

Herança de recursos

`Laminas\Permissions\Acl` fornece uma estrutura em árvore para a qual múltiplos recursos podem ser adicionados. Consultas sobre um recurso específico vão automaticamente procurar na hierarquia de

recursos por regras associadas aos recursos pais, permitindo herança simples de regras.

De modo coerente, se uma regra padrão deve ser aplicada a todos os recursos dentro de um tronco, é mais fácil associar as regras ao pai. A associação de exceções para aqueles recursos que não foram incluídos na regra pai pode ser facilmente imposta via `Laminas\Permissions\Acl`.

Note que um recurso pode herdar somente de um recurso pai, embora o pai possa fazer o mesmo com seu próprio pai, e assim por diante. A herança de recurso é definida pelo segundo argumento do método `addResource()` de `Laminas\Permissions\Acl\Acl`.

Herança de papéis

Ao contrário dos recursos, em `Laminas\Permissions\Acl` um papel pode herdar de um ou mais papéis para suportar a herança de regras. Enquanto essa habilidade pode ser extremamente útil em alguns momentos, ela também adiciona complexidade à herança. No caso de herança múltipla, se um conflito tem lugar entre as regras, a ordem na qual os papéis aparecem determina a herança final – a primeira regra encontrada via consulta é imposta.

O código seguinte define três papéis básicos – `convidado`, `membro` e `administrador` – dos quais outros papéis podem herdar. Então um papel identificado como `algumUsuario` é estabelecido e herda dos três outros. A ordem na qual esses papéis aparecem no array `$parents` é importante.

Quando necessário, `Laminas\Permissions\Acl` busca por regras de acesso definidas não somente pelo papel consultado (neste caso, `algumUsuario`), mas também pelos papéis dos quais o papel consultado herda (neste caso, `convidado`, `membro` e `administrador`).

```php
<?php
use Laminas\Permissions\Acl\Resource\GenericResource;
use Laminas\Permissions\Acl\Role\GenericRole;
use Laminas\Permissions\Acl\Acl;
$acl = new Acl();
$acl->addRole(new GenericRole('convidado'))
->addRole(new GenericRole('membro'))
->addRole(new GenericRole('administrador'));
$parents = ['convidado', 'membro', 'administrador'];
$acl->addRole(new GenericRole('algumUsuario'), $parents);
$acl->addResource(new GenericResource('algumRecurso'));
$acl->deny('convidado', 'algumRecurso');
$acl->allow('membro', 'algumRecurso');
echo $acl->isAllowed('algumUsuario', 'algumRecurso') ? 'permitido' :
'negado';
```

Definindo privilégios

A definição simples de privilégio é feita com o terceiro argumento dos métodos `allow()` e `deny()` de `Laminas\Permissions\Acl\Acl`.

Por exemplo, para definir os privilégios de visualização para todos os recursos da lista pelo papel `administrador`, podemos usar o seguinte código:

```php
$acl->allow('administrador', null,
['aprovar','criar','editar','revisar','ver']);
```

Para definir um único privilégio, pode ser usada uma string em vez de um array:

```php
$acl->allow('convidado', null, 'ver');
```

Associando regras via asserções

Há momentos em que uma regra não deve ser absoluta – quando o acesso para um recurso por um papel deve depender de vários critérios. `Laminas\Permissions\Acl` tem suporte embutido para

implementar regras baseado em condições que precisam ser satisfeitas, com o uso de `Laminas\Permissions\Acl\Assertion\AssertionInterface` e seu método `assert()`.

Uma vez que a classe de asserção tenha sido criada, uma instância da classe de asserção deve ser fornecida quando associar regras condicionais para uma ACL com o método `allow()` ou `deny()`. Uma regra assim criada será imposta somente quando o método de asserção retornar `TRUE`.

Você não entendeu nada? É o seguinte: você quer saber se um papel pode ter acesso a um recurso e quer estabelecer uma série de condições para isso. Onde você define as regras?

Em uma classe que estende `Laminas\Permissions\Acl\Assertion\AssertionInterface`. Vamos dar um exemplo:

```
<?php
namespace Muppet \Permissions\Acl\Assertion;
use Laminas\Permissions\Acl\Acl ;
use Laminas\Permissions\Acl\Role\RoleInterface ;
use Laminas\Permissions\Acl\Resource\ResourceInterface ;
use Laminas\Permissions\Acl\Assertion\AssertionInterface ;
class CleanIPAssertion implements AssertionInterface {
    public function assert(Acl $acl, RoleInterface $role = null,
ResourceInterface $resource = null, $privilege = null) {
        return $this->isCleanIP($_SERVER['REMOTE_ADDR']);
}
    protected function isCleanIP($ip) {
        // ...
    }
}
```

Uma vez que a classe de asserção (como a definida anteriormente) está disponível, o desenvolvedor deve fornecer uma instância da classe quando associar regras condicionais. Uma regra que é criada com uma asserção somente é aplicada quando o método retorna

`TRUE` . Uma asserção é uma afirmação. Quem vai verificar se ela é verdadeira ou não é... Você. Ou, melhor, o método `assert()` que você definiu. Como é, na prática? Podemos, por exemplo, criar uma lista de controle de acesso e definirmos as permissões gerais dessa lista:

```
<?php
use Muppet\Permissions\Acl\Assertion\CleanIPAssertion ;
use Laminas\Permissions\Acl\Acl;
$acl = new Acl();
$acl->allow(null, null, null, new CleanIPAssertion());
```

O código anterior cria uma regra de permissão condicional que permite acesso a todos os privilégios sobre todos os recursos para todos, exceto aqueles cujo IP requisitante estiver na "lista negra".

11.2 Laminas\Permissions\Rbac

O componente `Laminas\Permissions\Rbac` provê uma implementação de RBAC – Role Based Access Control (controle de acesso baseado em papéis) construída sobre as interfaces `RecursiveIterator` e `RecursiveIteratorIterator` da SPL, a biblioteca padrão de classes e interfaces do PHP. RBAC difere de ACL por dar mais ênfase aos papéis e às suas permissões do que aos objetos (recursos).

Como foi para ACL, algumas definições são necessárias antes de seguirmos em frente. Para RBAC:

- uma identidade tem um ou mais papéis;
- um papel requisita acesso a uma permissão;
- uma permissão é dada a um papel.

Ou seja:

- há um relacionamento de muitos para muitos entre identidades e papéis;

- há um relacionamento de muitos para muitos entre papéis e permissões;
- papéis podem ter um papel pai.

O recurso, como objeto, não existe na implementação RBAC. Também não damos permissões a partir do objeto RBAC. A permissão é dada a partir do objeto papel.

Criando um RBAC

Agora mostraremos como criar um RBAC com `Laminas\Permissions\Rbac`. No trecho de código a seguir instanciamos a classe que encapsulará os papéis e as permissões associadas a eles.

```
<?php
use Laminas\Permissions\Rbac\Role;
$rbac = new Rbac();
```

Criando papéis

Mostraremos aqui como criar uma classe que representa um papel. A classe precisa apenas estender `Laminas\Permissions\Rbac\AbstractRole`, mas `Laminas\Permissions\Rbac\Role` é fornecida como uma implementação básica. No trecho de código a seguir criamos dois papéis com a classe `Role`.

```
Laminas\Permission\Rbac\AbstractRole .
<?php
use Laminas\Permissions\Rbac\Role;
$role1 = new Role('chefe');
$role2 = new Role('peao');
```

Herança de papéis

Como em ACL, um papel RBAC pode herdar de um ou mais papéis. Você já viu esse filme, por isso, se sentir um *déjà vu*, é uma

sensação fundamentada. O código seguinte define três papéis básicos – `convidado`, `membro` e `administrador` – dos quais outros papéis podem herdar. Então, um papel identificado como `algumUsuario` é estabelecido e herda dos três outros papéis. A ordem na qual esses papéis aparecem no array `$parents` é importante. Quando necessário, `Laminas\Permissions\Rbac` busca por regras de acesso definidas não somente pelo papel consultado (neste caso, `algumUsuario`), mas também pelos papéis dos quais o papel consultado herda (neste caso, `convidado`, `membro` e `administrador`).

```
<?php
use Laminas\Permissions\Rbac\Role;
use Laminas\Permissions\Rbac\Rbac;
$rbac = new Rbac();
$role = new Role('convidado');
$role->addPermission('algumRecurso');
$rbac->addRole($role)
->addRole(new Role('membro'))
->addRole(new Role('administrador'));
$parents = ['convidado', 'membro', 'administrador'];
$rbac->addRole(new Role('algumUsuario'), $parents);
echo $rbac->isGranted('algumUsuario', 'algumRecurso') ? 'permitido' :
'negado';
```

Com isso, encerramos a apresentação das implementações do Laminas para duas estratégias de controle de permissões.

Capítulo 12
Mapeamento objeto-relacional com Laminas\Db

Aquele que tudo julga fácil encontrará muitas dificuldades. – Lao-Tsé

Neste capítulo, conheceremos mais detalhes sobre a implementação de abstração de banco de dados e mapeamento objeto-relacional do componente `Laminas\Db`.

12.1 Laminas\Db

O `Laminas\Db` já é utilizado pelo projeto corps3. O objetivo aqui é acrescentar conhecimento sobre ele. Esse componente é fracamente acoplado em relação à implementação MVC do Laminas, de modo que você pode utilizá-lo em qualquer aplicação PHP de modo independente.

A conexão com banco de dados no Laminas é feita com uma classe adaptadora, que torna possível evitar o acoplamento da aplicação com a sintaxe SQL e funções específicas de um determinado banco de dados. Use a classe `Laminas\Db\Adapter\Adapter` para preparar uma conexão. A conexão só será feita realmente quando a primeira operação de banco for realizada, evitando conectar sem necessidade.

O trecho a seguir mostra um exemplo de criação de uma instância de um adaptador.

```
<?php
use Laminas\Db\Adapter\Adapter;
$config = [
    'driver' => $driver,
```

```
    'database' => $database,
    'username' => $username,
    'password' => $password
];
$adapter = new Adapter($config);
```

A tabela a seguir mostra as chaves obrigatórias e opcionais do array passado como argumento para o construtor de `Adapter`.

Chave	Requerida	Valor
driver	Sempre	`Mysqli`, `Sqlsrv`, `Pdo_Sqlite`, `Pdo_Mysql`, `Pdo=OutroDriverPdo`
database	Geralmente	Nome do banco de dados
username	Geralmente	Usuário da conexão
password	Geralmente	Senha da conexão
hostname	Nem sempre	Endereço IP ou nome do servidor
port	Nem sempre	Porta de conexão
charset	Nem sempre	Codificação de caracteres utilizada

Executando comandos a partir do adaptador

A classe adaptadora pode executar comandos com o método `query()`. Para fazer consultas, os argumentos devem ser indicados por sinais de interrogação, que serão substituídos por valores de um array. O trecho de código a seguir mostra um exemplo de uma consulta.

```
$adapter->query('SELECT * FROM `lanterna` WHERE `codigo` = ?', [5]);
```

Quando executa uma consulta, o método `query()` retorna um objeto `Laminas\Db\ResultSet\ResultSet`. Esse objeto é iterável, o que significa que você pode usá-lo em uma estrutura `foreach`.

Para executar outros comandos, o segundo argumento do método `query()` deve receber como valor a constante `Adapter::QUERY_MODE_EXECUTE`. O trecho de código a seguir mostra um exemplo de execução de um comando para adicionar um índice a uma tabela.

```
$adapter->query('ALTER TABLE ADD COLUMN `bateria` int(11) NOT NULL',
Adapter::QUERY_MODE_EXECUTE);
```

Nesse caso, o método `query()` retorna um objeto de uma classe que implementa a interface `Laminas\Db\Adapter\Driver\StatementInterface`.

O método `query()` executa diretamente o comando recebido. Se você quiser preparar um comando SQL para ser executado posteriormente, deve utilizar o método `createStatement()`. A seguir temos um exemplo que tem o mesmo resultado final de `query()` com um comando que não é uma consulta.

```
$sql = 'ALTER TABLE `lanterna` ADD COLUMN `bateria` int(11) NOT NULL';
$statement = $adapter->createStatement($sql);
$result = $statement->execute();
```

O método `createStatement()` admite um segundo argumento, um objeto da classe que permite passar parâmetros opcionais que serão associados ao comando SQL para sua execução. `ParameterContainer` recebe um array como argumento em seu construtor, no qual você define os parâmetros.

Sintaxe SQL específica de banco

O problema de poder escrever um texto SQL diretamente para os métodos `query()` e `createStatement()` é que acabamos usando uma sintaxe específica de uma marca de banco de dados. É o que

ocorreu no tópico anterior, quando usamos caracteres delimitadores que funcionam para MySQL, mas não para PostgreSQL.

Você pode abstrair esses detalhes usando um objeto que os contenha. Esse objeto pode ser obtido com o método `getPlatform()` de `Adapter` . A instância do objeto que encapsula a sintaxe específica da plataforma fornece os seguintes métodos:

Método	Descrição
`getIdentifierSeparator()`	Retorna o caractere separador de identificadores.
`getQuoteIdentifierSymbol()`	Retorna o caractere delimitador de identificadores.
`getQuoteValueSymbol()`	Retorna o argumento com o delimitador de valores.
`quoteIdentifier()`	Retorna o argumento cercado pelo delimitador de identificadores (campos, por exemplo).
`quoteIdentifierChain()`	Retorna identificadores a partir de um array, delimitados e separados. Por exemplo, `array('schema','mytable')` torna-se `"schema"."table"\` (em PostgreSQL).
`quoteIdentifierInFragment()`	Retorna um fragmento de SQL com os identificadores delimitados. Opcionalmente, aceita um array como segundo argumento com palavras que não serão delimitadas.
`quoteValue()`	Retorna o argumento cercado pelo delimitador de valores.
`quoteValueList()`	Retorna valores a partir de um array, delimitados e separados por vírgula. Identificadores contidos nos valores são neutralizados.

Esses métodos também são extremamente úteis para evitar injeção de SQL.

Criação de comandos SQL

Você não precisa escrever comandos SQL se utilizar `Laminas\Db\Sql\Sql`. A instância dessa classe exige a injeção de um objeto `Adapter` em seu construtor. O objeto SQL permite criar qualquer comando DML (*Data Manipulation Language*) da SQL: `select`, `insert`, `update` e `delete`. O texto SQL de cada um desses comandos é encapsulado, respectivamente, pelas classes `Select`, `Insert`, `Update` e `Delete`.

A seguir, temos um exemplo de como declarar, preparar e executar um comando SQL `SELECT` com `Laminas\Db\Sql\Sql`. A variável `$adapter` é uma instância de `Laminas\Db\Adapter\Adapter`.

```
use Laminas\Db\Sql\Sql;
$sql = new Sql($adapter);
$select = $sql->select();
$select->from('lanterna')
->where(array('codigo' => 2));
$statement = $sql->prepareStatementForSqlObject($select);
$results = $statement->execute();
```

Para obter o texto SQL encapsulado pelo objeto de qualquer uma das quatro classes DML, usamos o método `getSqlStringForSqlObject()` de SQL, que precisa do objeto com os dados da plataforma para saber qual o dialeto de SQL. O exemplo a seguir mostra como fazer isso:

```
use Laminas\Db\Sql\Sql;
$sql = new Sql($adapter);
$select = $sql->select();
$select->from('lanterna')
->where(['codigo' => 2]);
$sqlString = $select->getSqlString($adapter)->getPlatform();
```

SELECT

São métodos da classe `Select`:

Método	Argumentos
`__construct()`	`$table = null`
`from()`	`$table`
`columns()`	`array $columns,$prefixColumnsWithTable = true`
`join()`	`$name, $on, $columns = self::SQL_STAR, $type = self::JOIN_INNER`
`where()`	`$predicate, $combination = Predicate\PredicateSet::OP_AND`
`group()`	`$group`
`having()`	`$predicate, $combination = Predicate\PredicateSet::OP_AND`
`order()`	`$order`
`limit()`	`$limit`
`offset()`	`$offset`

NOTA O nome da tabela pode ser fornecido por `__construct()` ou `from()`. As combinações dos predicados dos métodos `where()` e `having()` por padrão são conjunções (condições ligadas pelo operador `&&` ou `and`). Você pode modificar isso passando um segundo argumento para esses métodos, indicando outra forma de combinação.

Usamos o método `where()` nos projetos deste livro da forma mais básica possível, passando uma condição de igualdade, que é expressa por um array associativo. Mas o método `where()` pode receber como argumento um objeto `Laminas\Db\Sql\Where`. Esse objeto tem os seguintes métodos:

Método	Argumentos
equalTo	$left, $right, $leftType = self::TYPE_IDENTIFIER, $rightType = self::TYPE_VALUE
lessThan	$left, $right, $leftType = self::TYPE_IDENTIFIER, $rightType = self::TYPE_VALUE
greaterThan	$left, $right, $leftType = self::TYPE_IDENTIFIER, $rightType = self::TYPE_VALUE
lessThanOrEqualTo	$left, $right, $leftType = self::TYPE_IDENTIFIER, $rightType = self::TYPE_VALUE
greaterThanOrEqualTo	$left, $right, $leftType = self::TYPE_IDENTIFIER, $rightType = self::TYPE_VALUE
like	$identifier, $like
literal	$literal, $parameter
isNull	$identifier
isNotNull	$identifier
in	$identifier, array $valueSet = array()
between	$identifier, $minValue, $maxValue

INSERT

São métodos da classe `Insert` :

Método	Argumentos
__construct()	$table = null

Método	Argumentos
table()	$table
set()	array $values, $flag = self::VALUES_SET
where()	$predicate, $combination = Predicate\PredicateSet::OP_AND

UPDATE

São métodos da classe `Update` :

Método	Argumentos
__construct()	$table = null
table()	$table
set()	array $values, $flag = self::VALUES_SET
where()	$predicate, $combination = Predicate\PredicateSet::OP_AND

DELETE

São métodos da classe `Delete` :

Método	Argumentos
__construct()	$table = null
from()	$table
where()	$predicate, $combination = Predicate\PredicateSet::OP_AND

Mapeamento de tabelas

`Laminas\Db\TableGateway` é uma classe que permite mapear os dados de uma tabela. Ela é a ponte entre a entidade ou o modelo da aplicação e a tabela do banco de dados. São métodos da classe `Laminas\Db\TableGateway` :

Método	Argumentos ou efeitos
`__construct()`	`$table, Adapter $adapter, $features = null, ResultSet $resultSetPrototype = null, Sql $sql = null`
`getTable()`	Retorna o nome da tabela, fornecido pelo construtor.
`getAdapter()`	Retorna a instância de `Laminas\Db\Adapter\Adapter`.
`getColumns()`	Retorna os campos da tabela.
`getFeatureSet()`	Retorna a instância de `Laminas\Db\TableGateway\Feature\FeatureSet`.
`getResultSetPrototype`	Retorna a instância da classe que implementa `Laminas\Db\Resultset\ResultSetInterface`.
`getSql`	Retorna a instância de `Laminas\Db\Sql\Sql`.
`select()`	Faz uma consulta utilizando `$where` como filtro.
`selectWith()`	Faz uma consulta utilizando um objeto `Select` customizado.
`insert()`	Faz uma inclusão utilizando os dados do array `$set`.
`insertWith()`	Faz uma inclusão utilizando um objeto `Insert` customizado.
`update()`	Faz uma atualização utilizando os dados do array `$set` com `$where` como filtro.
`updateWith()`	Faz uma atualização utilizando um objeto `Update` customizado.

Método	Argumentos ou efeitos
`delete()`	Faz uma remoção utilizando `$where` como filtro.
`deleteWith()`	Faz uma remoção utilizando um objeto `Delete` customizado.

O argumento `$resultSetPrototype` define qual classe será utilizada para encapsular o conjunto de registros resultantes de uma consulta à tabela em uma coleção de objetos.

A classe `Laminas\Db\ResultSet\ResultSet` é a mais simples de utilizar para definir o conjunto de resultados. Ela exige apenas que o modelo implemente o método `exchangeArray()` para que ela possa popular os atributos dos objetos da coleção.

Porém, também existe a classe `Laminas\Db\ResultSet\HydratingResultSet`, que permite definir uma estratégia mais apropriada para lançar os dados de um registro para os atributos de um objeto.

Mapeamento de registros

Como vimos anteriormente, o argumento `$resultSetPrototype` define uma classe que representa o conjunto de registros. A classe `ResultSet` possui um método `setArrayObjectPrototype()`, que permite definir qual classe será utilizada para encapsular cada elemento da coleção, ou cada registro.

Para encapsular os registros, podemos utilizar uma classe PHP normal e depender do mapeador para fazer as operações de inclusão, alteração e remoção, ou ter um objeto persistente, que saiba como operar com os dados que contém.

A segunda opção pode ser implementada com a classe `Laminas\Db\RowGateway\RowGateway`. São métodos da classe `Laminas\Db\RowGateway\RowGateway`:

Método	Argumentos ou efeitos
`__construct()`	`$primaryKeyColumn, $table, $adapterOrSql = null`
`populate()`	Recebe um array `$rowData` com os dados do registro e uma `$rowExistsInDatabase` para indicar que já existe um registro com os dados.
`save()`	Se o registro existe, atualiza-o. Se não existe, cria-o.
`delete()`	Apaga o registro na tabela.

Mais adiante vamos aplicar `RowGateway` em uma cópia do projeto corps3 para compreender sem qualquer sombra de dúvida como ela pode ser utilizada.

Metadados de tabelas

Você precisa de informações sobre tabelas, visões, restrições e gatilhos? Instancie a classe `Laminas\Db\Metadata\Metadata`, injetando a instância de Adapter via construtor. São métodos da classe `Laminas\Db\Metadata\Metadata`:

Método	Argumentos
getTableNames	$schema = null, $includeViews = false
getTables	$schema = null, $includeViews = false
getTable	$tableName, $schema = null
getViewNames	$schema = null
getViews	$schema = null
getView	$viewName, $schema = null
getColumnNames	$table, $schema = null
getColumns	$table, $schema = null
getColumn	$columnName, $table, $schema = null
getConstraints	$table, $schema = null
getConstraint	$constraintName, $table, $schema = null
getConstraintKeys	$constraint, $table, $schema = null
getTriggerNames	$schema = null
getTriggers	$schema = null
getTrigger	$triggerName, $schema = null

Profiler

Precisa de informações sobre as consultas executadas? Você pode usar um profiler. A classe `Adapter` permite que você configure, com o método `setProfiler()`, um objeto profiler que armazenará informações sobre as consultas a partir do momento em que for iniciado. Laminas vem com a classe `Laminas\Db\Adapter\Profiler\Profiler` para esse fim. O objeto profiler de um adaptador de banco de dados pode ser recuperado com o

método `getProfiler()`. São métodos da classe `Laminas\Db\Adapter\Profiler\Profiler`:

Método	Argumentos
`profilerStart`	string|StatementContainerInterface $target
`profilerFinish`	Não tem.
`getLastProfile`	Não tem.
`getProfile`	Não tem.

12.2 Criando um projeto com o ORM do Laminas/Db

Copie o projeto corps3 como corps4.

Estendendo Laminas\Db\RowGateway\RowGateway

Nossos modelos estendem um modelo abstrato que define um comportamento genérico, mas ainda são limitados a meros repositórios de dados. As operações dependem completamente da implementação de Table Data Gateway. Vamos tornar os modelos mais poderosos, fazendo com que eles estendam a classe `RowGateway` de Laminas. Mas, com grandes poderes, vêm grandes responsabilidades. Nesse caso, as responsabilidades de `AbstractTableGateway` passarão para `AbstractModel` e isso afetará a relação de ambas com o controlador.

Segundo Fowler (2006, p. 158), o padrão de projeto Row Data Gateway "fornece objetos que parecem exatamente com o registro na sua estrutura de registros, mas que podem ser acessados com os mecanismos normais da sua linguagem de programação. Todos os detalhes de acesso à fonte de dados ficam escondidos atrás desta interface".

Vamos criar uma classe que herde da classe `AbstractModel` do framework Fgsl (usado no capítulo sobre autenticação) e sobrescreva o método `populate()` para indicar como saber se um objeto é transiente (novo, só existe na memória) ou persistente (existe no banco de dados). Vamos criar a classe `AbstractCustomModel` no diretório `src\Model` do módulo `Tropa`, assim:

```
<?php
namespace Tropa\Model;
use Fgsl\Model\AbstractModel;
use Laminas\InputFilter\InputFilterInterface;

abstract class AbstractCustomModel extends AbstractModel
{
    protected ?InputFilterInterface $inputFilter = null;

    abstract public function getInputFilter(): InputFilterInterface;

    public function populate($rowData, $rowExistsInDatabase = false)
    {
        if (isset($rowData['submit'])) unset($rowData['submit']);
        $select = $this->sql->select()->where([
            $this->primaryKeyColumn[0] =>
            $rowData[$this->primaryKeyColumn[0]]]);
        $statement = $this->sql->prepareStatementForSqlObject($select);
        $result = $statement->execute();
        $rowExistsInDatabase = ($result->count() > 0);
        parent::populate($rowData, $rowExistsInDatabase);
    }

    public function getArrayCopy()
    {
        return $this->data;
    }
}
```

O método `populate()` é sobrescrito para evitar que o botão de submissão do formulário seja considerado como campo da tabela.

Enxugando os modelos

Como visto anteriormente, a herança de `RowGateway` — por meio de `AbstractModel` — modificou consideravelmente nosso modelo abstrato. Agora nossos modelos concretos também ficarão enxutos. Vamos primeiro alterar a classe `Setor`, fazendo com que herde `AbstractModel` e removendo os métodos herdados, de modo que ela fique assim, apenas com o método `getInputFilter()`:

```
<?php
namespace Tropa\Model;
use Fgsl\InputFilter\InputFilter;
use Laminas\Filter\ToInt;
use Laminas\Filter\StripTags;
use Laminas\Filter\StringTrim;
use Laminas\Validator\Between;
use Laminas\Validator\StringLength;
use Laminas\InputFilter\InputFilterInterface;

class Setor extends AbstractCustomModel
{
    public function getInputFilter(): InputFilterInterface
    {
        if (! $this->inputFilter) {
            $inputFilter = new InputFilter();
            $inputFilter->addFilter('codigo', new ToInt())
            ->addValidator('codigo', new Between([
                'min' => 0,
                'max' => 3600
            ]))
            ->addFilter('nome', new StripTags())
            ->addFilter('nome', new StringTrim())
            ->addValidator('nome', new StringLength([
                'encoding' => 'UTF-8',
                'min' => 2,
                'max' => 30
```

```
            ]));
            $this->inputFilter = $inputFilter;
        }
        return $this->inputFilter;
    }
}
```

Efetuamos em seguida as mesmas alterações na classe `Lanterna` com uma adição: o modelo `Lanterna` precisa sobrescrever o método `save()` para remover o objeto `Setor` encapsulado no atributo de dados para que o comando SQL `INSERT` contenha somente a chave estrangeira do setor.

```
<?php
namespace Tropa\Model;

use Fgsl\InputFilter\InputFilter;
use Laminas\Filter\ToInt;
use Laminas\Filter\StripTags;
use Laminas\Filter\StringTrim;
use Laminas\Validator\Digits;
use Laminas\Validator\StringLength;
use Laminas\InputFilter\InputFilterInterface;

class Lanterna extends AbstractCustomModel
{
    public function populate($rowData, $rowExistsInDatabase = false)
    {
        $nomeSetor = (isset($rowData['setor'])) ? $rowData['setor'] :
null;
        $rowData['setor'] = new Setor('codigo','setor',$this->sql-
>getAdapter());
        $rowData['setor']->codigo = (isset($rowData['codigo_setor'])) ?
$rowData['codigo_setor'] : null;
        $rowData['setor']->nome = $nomeSetor;
        parent::populate($rowData, $rowExistsInDatabase);
    }

    public function getInputFilter(): InputFilterInterface
```

```
    {
        if (! $this->inputFilter) {
            $inputFilter = new InputFilter();
            $inputFilter->addFilter('nome', new StripTags())
            ->addFilter('nome', new StringTrim())
            ->addValidator('nome', new StringLength([
                'encoding' => 'UTF-8',
                'min' => 2,
                'max' => 30
            ]))
            ->addFilter('codigo_setor', new ToInt())
            ->addValidator('codigo_setor', new Digits())
            ->addChains();
            $this->inputFilter = $inputFilter;
        }
        return $this->inputFilter;
    }
    public function save()
    {
        parent::save();
    }
}
```

Nossos modelos não precisam mais declarar os campos mapeados de tabelas como atributos porque `RowGateway` provê um único atributo `$this->data` para encapsulá-los.

Herdando um controlador abstrato mais poderoso

As classes herdeiras de `AbstractCustomModel` estão prontas para se comportarem como um `Active Record`. Elas só precisam de lógica de domínio, pois conhecimento de persistência elas já têm.

De acordo com Fowler (2006, p. 165), o padrão `Active Record` é implementado quando "um objeto carrega tanto dados quanto comportamento. Muitos desses dados são persistentes e precisam ser armazenados em um banco de dados". É justamente o caso de nossos modelos. Seus dados são persistentes. Mas até agora a persistência foi feita com o auxílio de `TableGateway`.

Vamos substituir a herança das classes controladoras, de modo que elas façam uso de um controlador abstrato que facilite o uso de `Active Record`. Vamos substituir a herança da classe `Laminas\Mvc\Controller\AbstractActionController` por `Fgsl\Mvc\Controller\AbstractCrudController`.

Esse controlador integra-se ao modelo, ao mapeador objeto-relacional e ao formulário, provendo os métodos de inclusão, alteração, exclusão e consulta de registros a partir da definição das classes relacionadas. Isso torna o controlador mais enxuto.

A classe `SetorController` ficará assim a após a troca da classe mãe e a exclusão do código que se torna desnecessário:

```
<?php
declare(strict_types=1);

namespace Tropa\Controller;
use Fgsl\Mvc\Controller\AbstractCrudController;
use Laminas\Form\Form;
use Tropa\Form\Setor as SetorForm;

class SetorController extends AbstractCrudController
{
    use MenuTrait;

    protected int $itemCountPerPage = 10;

    protected string $modelClass = 'Tropa\Model\Setor';

    protected string $route = 'tropa';

    protected string $tableClass = 'Tropa\Model\SetorTable';

    public function getForm(bool $full = FALSE): Form
    {
        return new SetorForm();
    }
```

```
    public function getEditTitle($key): string
    {
        return (is_null($key) ? 'Incluir' : 'Alterar') . ' Setor';
    }
}
```

A classe `LanternaController`, por sua vez, ficará assim:

```
<?php
declare(strict_types=1);

namespace Tropa\Controller;
use Fgsl\Mvc\Controller\AbstractCrudController;
use Laminas\Form\Form;
use Tropa\Form\Lanterna as LanternaForm;

class LanternaController extends AbstractCrudController
{
    use MenuTrait;

    protected int $itemCountPerPage = 10;

    protected string $modelClass = 'Tropa\Model\Lanterna';

    protected string $route = 'tropa';

    protected string $tableClass = 'Tropa\Model\LanternaTable';

    public function getForm(bool $full = FALSE): Form
    {
        return new LanternaForm(null,['table'=>$this->parentTable]);
    }

    public function getEditTitle($key): string
    {
        return (is_null($key) ? 'Incluir' : 'Alterar') . ' Lanterna';
    }
}
```

Observe que implementamos o método `getForm`, injetando a tabela de setores no formulário do cadastro de lanternas, porque o setor é uma chave estrangeira na tabela de lanternas e será exibido como uma caixa de seleção.

A classe `AbstractCrudController` tem um construtor que requer a injeção de dependências. Por isso, vamos criar fábricas para gerar as instâncias de nossos controladores.

Primeiro, vamos criar a classe `SetorControllerFactory`, para criar a instância de `SetorController`:

```
<?php
namespace Tropa\Controller;

use Laminas\ServiceManager\Factory\FactoryInterface;
use Psr\Container\ContainerInterface;
use Tropa\Model\SetorTable;
use Laminas\Db\TableGateway\Tablegateway;

class SetorControllerFactory implements FactoryInterface
{

    public function __invoke(ContainerInterface $container,
$requestedName, ?array $options = null): SetorController
    {
        $adapter = $container->get('LaminasDbAdapter');
        $table = new SetorTable(new TableGateway('setor',$adapter));
        return new SetorController($table);
    }
}
```

Na sequência, vamos criar a classe `LanternaControllerFactory`, para criar a instância de `LanternaController`:

```
<?php
namespace Tropa\Controller;

use Laminas\ServiceManager\Factory\FactoryInterface;
```

```
use Psr\Container\ContainerInterface;
use Tropa\Model\LanternaTable;
use Tropa\Model\SetorTable;
use Laminas\Db\TableGateway\Tablegateway;

class LanternaControllerFactory implements FactoryInterface
{

    public function __invoke(ContainerInterface $container,
$requestedName, ?array $options = null): LanternaController
    {
        $adapter = $container->get('LaminasDbAdapter');
        $table = new LanternaTable(new TableGateway('lanterna',$adapter));
        $parentTable = new SetorTable(new TableGateway('setor',$adapter));
        return new LanternaController($table, $parenTable);
    }
}
```

As fábricas não são invocadas automaticamente. Precisamos definir essa chamada no arquivo de configuração do módulo. Vamos modificar a chave `controllers => factories` para conectar as classes controladoras com suas respectivas fábricas, assim:

```
        'factories' => [
            Controller\SetorController::class =>
Controller\SetorControllerFactory::class,
            Controller\LanternaController::class =>
Controller\LanternaControllerFactory::class
        ],
```

12.3 Herdando uma classe mapeadora abstrata mais poderosa

Observe que as fábricas injetam instâncias das classes mapeadoras de tabelas. Essas classes encapsulam a classe `TableGateway` do Laminas. Mas a classe `AbstractCrudController` exige que as classes

injetadas sejam herdeiras de `Fgsl\Db\TableGateway\AbstractTableGateway`. Assim, teremos que modificar as classes `SetorTable` e `LanternaTable` para herdarem de `AbstractTableGateway` do framework `Fgsl`. Essa mudança enxugará as classes mapeadoras, pois a herança descartará a implementação da maioria dos métodos.

Vamos alterar primeiro a classe `SetorTable`:

```
<?php
namespace Tropa\Model;

use Fgsl\Db\TableGateway\AbstractTableGateway;

class SetorTable extends AbstractTableGateway
{
    protected string $keyName = 'codigo';

    protected string $modelName = 'Tropa\Model\Setor';
}
```

Depois, vamos alterar a classe `LanternaTable`:

```
<?php
namespace Tropa\Model;

use Fgsl\Db\TableGateway\AbstractTableGateway;;

class LanternaTable extends AbstractTableGateway
{
    protected string $keyName = 'codigo';

    protected string $modelName = 'Tropa\Model\Lanterna';
}
```

A classe `AbstractTableGateway` possui um método `getModel`, que retorna uma instância do modelo configurado pelo atributo `$modelName`. Mas há uma exigência: a classe modelo deve ser herdeira de `Fgsl\Model\AbstractActiveRecord`. Você pode pensar que

teremos alterar os dois modelos do módulo `Tropa`, `Setor` e `Lanterna`. Na verdade, não, pois esses modelos herdam de uma classe comum que criamos para o módulo, `AbstractCustomModel`.

A criação de uma classe intermediária que não acople diretamente as classes da aplicação com dependências de terceiros é uma boa estratégia, pois, caso precisemos alterar as dependências, reduzimos a quantidade de locais para fazer alterações. Assim, precisamos mudar apenas a implementação de `AbstractCustomModel`. Ela ficará assim:

```
<?php
namespace Tropa\Model;
use Fgsl\Model\AbstractActiveRecord;
use Laminas\InputFilter\InputFilterInterface;

abstract class AbstractCustomModel extends AbstractActiveRecord
{
}
```

Veja que removemos toda a implementação de `AbstractCustomModel`, porque `AbstractActiveRecord` implementa todos os métodos necessários. Você pode questionar a razão de ter uma classe que não tem implementação e apenas herda de outra. Como dissemos, esse é o ponto em que nos protegemos de mudanças de terceiros. Além disso, podemos adicionar no futuro às classes modelo funcionalidades que não estão no componente de terceiros. É como se pudéssemos modificar a classe externa herdada sem alterar diretamente o código-fonte dela.

Com essa mudança, nossas classes modelo se tornam descendentes de `AbstractActiveRecord` e podem ser usadas por `AbstractCrudController`.

Ajustando as coleções na criação dos mapeadores

As fábricas dos controladores precisam ser modificadas, para configurar os mapeadores a retornar coleções de objetos do tipo da

classe modelo relacionada. Se não definirmos qual o tipo de classe na qual os registros devem ser encapsulados, o mapeador retornará uma coleção de objetos da classe `ArrayObject` e isso provocará um erro nos métodos `getModel` e `getModels()` de `AbstractTableGateway`, que retornam uma instância de `AbstractActiveRecord`.

Vamos usar a classe `Laminas\Db\Result\Result` para definir o tipo das coleções retornadas pelas consultas feitas pelos mapeadores. Vamos começar alterando a classe `SetorControllerFactory`:

```
<?php
namespace Tropa\Controller;

use Laminas\ServiceManager\Factory\FactoryInterface;
use Psr\Container\ContainerInterface;
use Tropa\Model\SetorTable;
use Laminas\Db\TableGateway\TableGateway;
use Laminas\Db\ResultSet\ResultSet;
use Tropa\Model\Setor;

class SetorControllerFactory implements FactoryInterface
{

    public function __invoke(ContainerInterface $container,
$requestedName, ?array $options = null): SetorController
    {
        $adapter = $container->get('LaminasDbAdapter');
        $resultSetPrototype = new ResultSet();
        $resultSetPrototype->setArrayObjectPrototype(new Setor(
            'codigo',
            'setor',
            $adapter));
        $table = new SetorTable(new
TableGateway('setor',$adapter,null,$resultSetPrototype));
        return new SetorController($table);
    }
}
```

Em seguida, alteramos a classe `LanternaControllerFactory`:

```php
<?php
namespace Tropa\Controller;

use Laminas\ServiceManager\Factory\FactoryInterface;
use Psr\Container\ContainerInterface;
use Tropa\Model\LanternaTable;
use Tropa\Model\SetorTable;
use Laminas\Db\TableGateway\TableGateway;
use Laminas\Db\ResultSet\ResultSet;
use Tropa\Model\Lanterna;
use Tropa\Model\Setor;

class LanternaControllerFactory implements FactoryInterface
{

    public function __invoke(ContainerInterface $container,
$requestedName, ?array $options = null): LanternaController
    {
        $adapter = $container->get('LaminasDbAdapter');
        $resultSetPrototype = new ResultSet();
        $resultSetPrototype->setArrayObjectPrototype(new Lanterna(
            'codigo',
            'lanterna',
            $adapter));
        $table = new LanternaTable(new
TableGateway('lanterna',$adapter,null,$resultSetPrototype));
        $resultSetPrototype = new ResultSet();
        $resultSetPrototype->setArrayObjectPrototype(new Setor(
            'codigo',
            'setor',
            $adapter));
        $parentTable = new SetorTable(new
TableGateway('setor',$adapter,null,$resultSetPrototype));
        return new LanternaController($table, $parentTable);
    }
}
```

12.4 Gerenciando relacionamentos na persistência de dados

O modelo `Lanterna` depende do modelo `Setor`. O código do setor é uma chave estrangeira na tabela de lanternas. No modelo de objetos da aplicação, o objeto da classe `Lanterna` tem um atributo que é um objeto da classe `Setor`. Isso é conveniente para a aplicação, pois os dados relacionados estão contidos em um único elemento, mas na hora de gravar na tabela do banco de dados um registro da tabela de lanternas, não gravamos o nome do setor, porque esse é um campo de outra tabela. Nós gravamos apenas a chave estrangeira.

Assim, na hora de gravar o registro, o modelo e o mapeador objeto-relacional precisam extrair a chave estrangeira do objeto relacionado e passar apenas os dados que os comandos SQL `INSERT` e `UPDATE` precisam.

Primeiro, vamos alterar o método `getArrayCopy()` da classe `Lanterna`. Esse método é herdado da classe avó (`AbstractTableGateway`) e retorna os valores dos atributos do objeto na forma de um array.

```
public function getArrayCopy()
{
    $data = $this->data;
    $data['codigo_setor'] = $data['setor']->codigo;
    unset($data['setor']);
    return $data;
}
```

Com essa alteração, o modelo `Lanterna` retorna um array com os valores aceitos como válidos para inclusão ou alteração na tabela de lanternas.

Em seguida, vamos alterar o método `save()` da classe `LanternaTable`, para que exclua a chave primária na gravação. Ao contrário da tabela de setores, onde o código é fornecido pelo usuário, o código da tabela lanterna é gerado automaticamente, então não deve ser

incluído no comando SQL como parte dos valores a serem gravados. Ele só deve ser usado como filtro para alteração de registros.

Vamos sobrescrever o método, alterando o valor do segundo argumento, opcional, para que exclua a chave primária do conjunto de campos a serem gravados.

```
    public function save(AbstractActiveRecord $model, $excludePrimaryKey =
false)
    {
        parent::save($model,true);
    }
```

Com isso, o projeto `corps4` torna-se funcional, utilizando `RowGateway`, por meio de extensões, e assim implementamos um mapeamento objeto-relacional completo com o componente `Laminas\Db`.

Capítulo 13
Web services e APIs

Se você tem exposto tudo, você não pode mudar alguma coisa ou você quebra todos os seus clientes. É por isso que o modelo de componente, o foco na API, projetando o que é interno e o que é publicado, torna-se tão crítico quando vem para o reúso. – Erich Gamma

Neste capítulo, aprenderemos como implementar web services em PHP com Laminas usando os protocolos XML-RPC, SOAP e JSON-RPC. Este não é um livro sobre web services, por isso apenas veremos como criar e consumir serviços com esses protocolos, sem nos deter nas explicações.

13.1 XML-RPC

XML-RPC (*Extensible Markup Language — Remote Procedure Call*) é uma especificação e um conjunto de implementações que permitem que softwares executados em sistemas operacionais distintos, em ambientes diferentes, façam chamadas de procedimentos pela Internet usando XML como padrão de troca de mensagens. XML-RPC é uma especificação criada por Dave Winner, que está documentada em http://xmlrpc.com/spec.md.

Como criar um cliente de um web service XML-RPC

Primeiro, crie uma instância da classe `Laminas\XmlRpc\Client`. No construtor dessa classe, passe o URL do servidor.

```
$client = new Laminas\XmlRpc\Client($url);
```

Depois, invoque os serviços utilizando o método `call()`. Esse método recebe como argumento um texto no formato `namespace.method`. Por exemplo:

```
$comercial = $client->call('bancocentral.dolarComercial');
```

O provedor do serviço tem que informar como o método deve ser consumido, é claro. Neste caso estamos chamando um serviço XML-RPC que é construído com `Laminas\XmlRpc` também. Na próxima seção você verá como um método de classe é transformado em serviço web.

Para passar argumentos para o serviço (que deve ser, em sua fonte, uma função ou um método de classe), use o segundo argumento de `call()`, que recebe um array.

Se você quiser manipular os serviços como se fossem objetos locais, pode utilizar um proxy. O proxy é obtido pelo método `getProxy()`. A partir dele, os métodos do objeto remoto podem ser chamados como se fossem locais. Por exemplo:

```
$service = client->getProxy('bancocentral');
$comercial = $service->dolarComercial();
$paralelo = $service->dolar('paralelo');
```

Como saber quais serviços estão disponíveis

O método `getIntrospector()` de `Laminas\XmlRpc\Client` retorna um objeto que permite obter os serviços disponíveis. O objeto da classe `Laminas\Xml\Client\ServerIntrospection` possui um método `listMethods()` que retorna todos os métodos disponíveis no URL apontado pelo cliente.

Exceções de Laminas\XmlRpc\Client

Existem duas exceções para o cliente de XML-RPC. A primeira refere-se a falhas decorrentes da comunicação utilizando o protocolo HTTP. A segunda refere-se a falhas decorrentes da comunicação utilizando o protocolo XML-RPC. São elas, respectivamente:

- `Laminas\XmlRpc\Client\Exception\HttpException`
- `Laminas\XmlRpc\Client\Exception\FaultException`

Como criar um servidor de web services XML-RPC

Primeiro, crie uma instância da classe `Laminas\XmlRpc\Server`.

```
$server = new Laminas\XmlRpc\Server();
```

Depois, crie os serviços, adicionando classes ou funções ao servidor. Para adicionar classes, use o método `setClass()`, que recebe como argumentos o namespace da classe e o namespace do serviço. Por exemplo:

```
$server->setClass('Service\Public\BancoCentral','bancocentral');
```

Para adicionar funções, use o método `addFunction()`, que recebe como argumentos o nome da função e o namespace do serviço. Por exemplo:

```
$server->addFunction('calcularParalaxe','astronomia');
```

Os serviços são expostos pelo método `handle()`. Por exemplo:

```
echo $server->handle();
```

Como transformar métodos de uma classe ou funções em serviços XML-RPC

Para que `Laminas\XmlRpc\Server` gere os descritores de serviço, é necessário que os métodos e funções sejam anotados com blocos de documentação PHPDoc. Os blocos devem ter as anotações `@param` e `@return`. Você pode utilizar tipos PHP ou XML-RPC para descrever o

tipo utilizado pelos parâmetros e argumentos. A tabela a seguir descreve os tipos PHP e XML-RPC e sua correspondência.

Tipo nativo PHP	Tipo XML-RPC
integer	int
Laminas\Math\BigInteger\BigInteger	i8
double	double
boolean	boolean
string	string
null	nil
array	array
associative	array struct
object	array
DateTime	dateTime.iso8601

13.2 SOAP

SOAP (*Simple Object Access Protocol*) é outro protocolo de troca de mensagens entre aplicações usando XML. SOAP é uma especificacação do W3C, que está disponível em www.w3.org/TR/SOAP.

Como criar um cliente de um web service SOAP

Primeiro, crie uma instância da classe `Laminas\Soap\Client`. No construtor dessa classe, passe o URL para o arquivo WSDL que contém a descrição dos serviços SOAP.

```
$client = new Laminas\Soap\Client($wsdl);
```

Você pode passar argumentos opcionais por meio de um segundo parâmetro do construtor, que é um array.

Os elementos aceitos por esse array são:

Opção	Significado
`soap_version`	Versão de SOAP a ser utilizada. A classe possui as constantes `SOAP_1_1` e `SOAP_1_2` .
`classmap`	Um array com tipos WSDL como chaves e nomes de classes PHP como valores.
`encoding`	Codificação interna de caracteres.
`wsdl`	Equivale a chamar `setWsdl()` ou passar o URL do WSDL pelo construtor.
`uri`	Namespace para o serviço SOAP, obrigatório para modo não WSDL.
`location`	URL para requisitar.
`style`	Estilo da requisição, só funciona no modo WSDL.
`use`	Método para codificar mensagens.
`login`	Usuário para autenticação HTTP.
`password`	Senha para autenticação HTTP.
`proxy_host`	Servidor de proxy.
`proxy_port`	Porta do serviço e proxy.
`proxy_login`	Usuário do Proxy.
`proxy_password`	Senha para o Proxy.
`local_cert`	Opção para autenticação com certificado.
`paraphrase`	Opção para autenticação com paráfrase (senha).

Opção	Significado
`compression`	Opção para compressão. Aceita as constantes `SOAP_COMPRESSION_ACCEPT`, `SOAP_COMPRESSION_GZIP` e `SOAP_COMPRESSION_DEFLATE`.

Os métodos das classes providas como serviços por `Laminas\Soap\Server` são chamados diretamente pelo objeto `Laminas\Soap\Client`, como se fossem dele. Por exemplo:

```
$comercial = $client->dolarComercial();
```

Como saber quais serviços estão disponíveis

SOAP utiliza um arquivo descritor de serviços. O método `toXML()` da classe `Laminas\Soap\AutoDiscover` provê o conteúdo desse arquivo. Veja o tópico adiante.

Como criar um servidor de web services SOAP

Para criar um servidor SOAP, é preciso criar o descritor de serviços primeiros. O código a seguir permite que o mesmo endereço forneça o arquivo WSDL e a resposta do servidor SOAP.

```
if (isset($_GET['wsdl'])) {
    $autodiscover = new Laminas\Soap\AutoDiscover();
    $autodiscover->setClass('BancoCentral')
        ->setUri('http://soapsample/server.php');
    echo $autodiscover->toXml();
} else {
    $server = new Laminas\Soap\Server("http://soapsample/server.php?
wsdl");
    $server->setClass('BancoCentral');
    $server->handle();
}
```

Para adicionar funções, use o método `addFunction()`. Por exemplo:

```
$server->addFunction('calcularParalaxe');
```

Como transformar métodos de uma classe ou funções em serviços SOAP

Para que `Laminas\Soap\Autodiscover` gere os descritores de serviço, é necessário que os métodos e as funções sejam anotados com blocos de documentação PHPDoc. Os blocos devem ter as anotações `@param` e `@return`.

13.3 JSON-RPC

JSON-RPC (*Javascript Object Notation — Remote Procedure Call*) é uma especificação de procoloco documentada em https://www.jsonrpc.org. Dos três componentes ofertados pelo Laminas, este representa a tendência contemporânea de comunicação entre aplicações em rede.

Como criar um servidor de web services JSON-RPC

Não falaremos sobre como criar um cliente para JSON-RPC pois esta é justamente a maior vantagem do protocolo JSON-RPC: a dispensa de uma formalização para consumir os dados de um servidor JSON-RPC.

As linguagens de programação mais utilizadas possuem suporte para ler e gravar no formato JSON, que é uma especificação de objeto na linguagem JavaScript. PHP implementa funções que convertem texto JSON em array e arrays em texto JSON. Isso torna fácil para aplicações PHP consumirem serviços escritos em JSON-RPC.

Para criar um servidor JSON-RPC, crie uma instância da classe `Laminas\Json\Server\Server`.

```
$server = new Laminas\Json\Server\Server();
```

Depois, crie os serviços, adicionando classes ou funções ao servidor. Para adicionar classes, use o método `setClass()`, que recebe como argumentos o namespace da classe e o namespace do serviço. Por exemplo:

```
$server->setClass('Service\Public\BancoCentral');
```

Para adicionar funções, use o método `addFunction()`, que recebe como argumentos o nome da função e o namespace do serviço. Por exemplo:

```
$server->addFunction('calcularParalaxe');
```

Os serviços são expostos pelo método `handle()`. Por exemplo:

```
echo $server->handle();
```

Como transformar métodos de uma classe ou funções em serviços JSON-RPC

Para que `Laminas\XmlRpc\Server` gere os descritores de serviço, é necessário que os métodos e funções sejam anotados com blocos de documentação PHPDoc. Os blocos devem ter as anotações `@param` e `@return`.

JSON-RPC é o protocolo padrão para serviços atuais, mas no mundo do software é preciso lidar com legado, por isso é importante saber como lidar com XML-RPC e SOAP, com os quais há muitos serviços implementados e em produção. Em uma eventual manutenção de serviços nesses protocolos, uma evolução gradual pode ser realizada, primeiro encapsulando a implementação atual nos respectivos componentes do Laminas e depois trocando as classes servidoras.

É claro que uma migração de web services envolve migrar não somente os servidores, mas também os clientes. Na verdade, é preciso manter os servidores com os protocolos atuais em funcionamento (XML-RPC e SOAP) enquanto se coloca um JSON em produção, porque os consumidores têm de implementar seus

clientes JSON primeiro para depois poderem migrar. E para os implementarem, eles precisam descobrir os serviços JSON disponíveis.

Capítulo 14
Lendo e gravando dados de configuração

A humildade é a base e o fundamento de todas as virtudes, e sem ela não há nenhuma que o seja. – Miguel de Cervantes

Neste capítulo, aprendemos a utilizar o componente `Laminas\Config`.

14.1 Laminas\Config

`Laminas\Config` foi projetado para simplificar o acesso e o uso de dados de configuração dentro de aplicações. Ele provê uma interface de usuário baseada em propriedades de objetos aninhadas para acessar os dados dentro de um código. Os dados de configuração podem vir de uma variedade de mídias que suportem armazenamento hierárquico de dados. Atualmente, `Laminas\Config` fornece adaptadores para dados de configuração que estão armazenados em arquivos de texto nos formatos .ini, JSON, YAML e XML.

Dados de configuração em um array PHP

Se os dados de configuração estiverem disponíveis em um array PHP, simplesmente passe os dados para o construtor de `Laminas\Config\Config` para utilizar uma interface simples orientada a objetos.

```
<?php
// Dado um array de dados de configuração
$configArray = array(
    'db' => array(
```

```
        'driver' => 'pdo_mysql',
        'host' => 'db.example.com',
        'username' => 'dbuser',
        'password' => 'secret',
        'dbname'    => 'mydatabase'
    )
);
// Cria um objeto de configuração
$config = new Laminas\Config\Config($configArray);
// Imprime um dado de configuração (resulta em 'db.example.com')
echo $config->db->host;
```

Se o array estiver dentro de um arquivo PHP, antecedido por um `return`, você pode importá-lo diretamente para o construtor de `Laminas\Config\Config`, assim:

```
$config = new Laminas\Config\Config(require $configFile);
```

Como a configuração funciona

Dados de configuração ficam acessíveis para o construtor de `Laminas\Config\Config` por meio de um array associativo, que pode ser multidimensional, de modo a suportar a organização de dados do geral para o específico.

Classes adaptadoras concretas são usadas no processo. Scripts de usuário podem fornecer tais arrays diretamente para o construtor de `Laminas\Config\Config`, sem usar uma classe adaptadora, sempre que for mais apropriado.

Cada valor de array de dados de configuração torna-se uma propriedade do objeto `Laminas\Config\Config`, com a chave usada como o nome de propriedade. Se um valor é ele próprio um array, então a propriedade do objeto resultante é um novo objeto `Laminas\Config\Config`, carregado com os dados do array. Isso ocorre recursivamente, de tal forma que uma hierarquia de dados de configuração pode ser criada com qualquer número de níveis.

`Laminas\Config\Config` implementa as interfaces `Countable` e `Iterator` de modo a facilitar o simples acesso aos dados de configuração, permitindo o uso da função `count()` e construções PHP tais como `foreach` sobre objetos `Laminas\Config\Config`.

Dois objetos `Laminas\Config\Config` podem ser mesclados em um único objeto com a função `merge()`.

Leitores de configuração

Para ler dados de configuração em formatos que não sejam PHP, existem classes leitoras específicas:

- `Laminas\Config\Reader\Ini`
- `Laminas\Config\Reader\Xml`
- `Laminas\Config\Reader\Json`
- `Laminas\Config\Reader\Yaml`
- `Laminas\Config\Reader\JavaProperties`

Todas essas classes implementam o método `fromFile()` que recebe o caminho para o arquivo de configuração a ser lido e retorna um array.

Escritores de configuração

Para converter um objeto `Laminas\Config\Config` para outro formato de dados, existem classes escritoras específicas:

- `Laminas\Config\Writer\Ini`
- `Laminas\Config\Writer\Xml`
- `Laminas\Config\Writer\PhpArray`
- `Laminas\Config\Writer\Json`
- `Laminas\Config\Writer\Yaml`
- `Laminas\Config\Writer\JavaProperties`

Todas essas classes implementam o método `toString()` que recebe um objeto `Laminas\Config\Config` e retorna o formato correspondente à classe (texto ou array).

CAPÍTULO 15
Internacionalização

Aprender várias línguas é questão de um ou dois anos; ser eloquente na sua própria exige a metade de uma vida.– Voltaire

Neste capítulo, aprendemos a utilizar o componente `Laminas\i18n` a fim de preparar nossas aplicações para que trabalhem com vários idiomas.

15.1 Laminas\I18n

Se você já teve a pretensão de ser poliglota, mas nunca teve a oportunidade de aprender mais de quatro línguas, essa é a sua chance de se realizar por intermédio da construção de programas que podem falar todas as línguas possíveis! Sem a necessidade de um tradutor universal do Professor Pardal ou um anel energético da Tropa dos Lanternas Verdes.

`Laminas\I18n` vem com um completo conjunto de classes de tradução que suporta a maioria dos formatos e inclui características populares como traduções plurais e domínios de texto. Se tudo der certo, ao fim deste capítulo, você deverá ser capaz de saber o suficiente sobre `Laminas\I18n` para traduzir textos, números, moedas e datas.

Tradução de textos

`Laminas\i18n\Translator\Translator` é uma classe para tradução geral de textos que depende da extensão PHP Intl. Você descobrirá facilmente se tem essa extensão ao tentar usar a classe `Translator` e obter um erro informando que a classe Locale não foi encontrada.

Para adicionar traduções a um objeto tradutor, há duas opções. Você pode adicionar todos os arquivos de tradução individualmente, o que é a melhor forma, se você usar formatos de tradução que armazenam várias localidades no mesmo arquivo, ou você pode adicionar traduções por meio de um padrão, o que funciona melhor para formatos que contêm uma localidade por arquivo.

Para adicionar um único arquivo para o tradutor, use o método `addTranslationFile()`:

```
use Laminas\I18n\Translator\Translator;
$translator = new Translator();
$translator->addTranslationFile($type, $filename, $textDomain, $locale);
```

O argumento `$type` pode assumir os valores `gettext`, `ini` ou `phpArray`, disponíveis no Laminas, ou qualquer identificador de uma classe que implemente a interface `FileLoaderInterface`.

O argumento `$textDomain` é opcional e representa um nome de categoria. Por padrão, vale "default". O argumento `$locale` é opcional; se não for informado, será utilizado o valor retornado por `Locale::getDefault()`.

Para adicionar um padrão para o tradutor, use o método `addTranslationFilePattern()`:

```
use Laminas\I18n\Translator\Translator;
$translator = new Translator();
$translator->addTranslationFilePattern($type, $pattern, $textDomain);
Para traduzir uma mensagem, usamos o método translate() de Translator :
$translator->translate($message, $textDomain, $locale);
```

Para traduzir singular ou plural, usamos o método `translate()` de `Translator`:

```
$translator->translatePlural($singular, $plural, $number, $textDomain,
$locale);
```

O argumento `$number` determina se será usado o singular ou plural. O número 1 é singular, qualquer outro valor é plural.

Existe uma *view helper* para tradução de textos nos arquivos `phtml`. Basta anexar o objeto `Translator` à *view helper* `Translate` para traduzir textos nas visões:

```
$this->translate($message, $textDomain, $locale);
```

Tradução de moedas

Conheceremos doravante algumas *view helpers* especificamente para tradução. Todas as chamadas a métodos deste e dos próximos tópicos deste capítulo estão no contexto dos arquivos `phtml`. `CurrencyFormat` é uma classe auxiliar para tradução de formatos de moedas para qualquer localidade. A chamada a `helper` é feita assim:

```
echo $this->currencyFormat($number, $currencyCode, $locale);
```

O valor de `$currencyCode` deve ser um código de três letras no padrão ISO 4217 (veja mais em http://en.wikipedia.org/wiki/ISO_4217/). O último argumento é opcional; se não for informado, será utilizado o valor retornado por `Locale::getDefault()`.

Tradução de datas

`DateFormat` é uma classe auxiliar para tradução de formatos de datas para qualquer localidade. A chamada a `helper` é feita assim:

```
echo $this->dateFormat(
    $date,
    $dateType,
    $timeType,
    $locale
);
```

Os três últimos argumentos são opcionais. `$date` é o valor de data a ser formatado. Pode ser uma instância de `DateTime`, um inteiro representando um timestamp Unix ou um array retornado por `localtime()`. `$dateType` e `$timeType` determinam respectivamente como a data e a hora serão representadas. Eles podem assumir o

valor de qualquer uma dessas constantes de `IntlDateFormatter`: `NONE` (padrão), `SHORT`, `MEDIUM`, `LONG` ou `FULL`. Se `$locale` não for informado, será utilizado o valor retornado por `Locale::getDefault()`.

Tradução de números

`NumberFormat` é uma classe auxiliar para tradução de formatos de números para qualquer localidade. A chamada a `helper` é feita assim:

```
echo $this->numberFormat(
    $number,
    $formatStyle,
    $formatType,
    $locale
);
```

Os três últimos argumentos são opcionais. `$formatStyle` usa por padrão o valor de `NumberFormatter::DECIMAL`. `$formatType` usa por padrão o valor de `NumberFormatter::TYPE_DEFAULT`. A classe `NumberFormatter` faz parte da extensão `Intl` do PHP. Se `$locale` não for informado, será utilizado o valor retornado por `Locale::getDefault()`.

Capítulo 16
Referências

DJIKSTRA, E. W. *The threats to computing science*. ACM South Central Regional Conference. Nov. 1984.

FOWLER, M. *Inversion of Control Containers and the Dependency Injection pattern*. 23 de janeiro de 2004. https://www.martinfowler.com/articles/injection.html

GAMMA, E. *Erich Gamma on Flexibility and Reuse*: A Conversation with Erich Gamma, Part II. Artima, Mai. 2005.

GAMMA, E. HELM, R. JOHNSON, R. VLISSIDES, J. *Design Patterns*: Elements of reusable object-oriented software. Addison-Wesley, 1995.

HOWARD, M. e LEBLANC, D. *Escrevendo código seguro*: estratégias e técnicas práticas para codificação segura de aplicativos em um mundo em rede. 2.ed. Porto Alegre: Bookman, 2005.

MCCONNELL, S. *Code Complete*. 2. ed. Microsoft Press, 2004.